Exploring High Performance Deep Neural Networks on GPUs

A Dissertation Presented
by

Shi Dong

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in

Computer Engineering

Northeastern University
Boston, Massachusetts

August 2020

To my beloved family.

Contents

List of Figures

List of Tables

List of Acronyms

AI Artificial Intelligence.

ANN Artificial Neural Network.

CNN Convolutional Neural Network.

DNN Deep Neural Network.

GEMM General Matrix Multiplication.

LSTM Long Short Term Memory.

ML Machine Learning.

RNN Recurrent Neural Network.

SGD Stochastic Gradient Descent.

CPU Central Processing Unit.

GPU Graphic Processing Unit.

TPU Tensor Processing Unit

FPGA Field-Programmable Gate Array

ASIC Application Specific Integrated Circuit

ISA Instruction Set Architecture.

CUDA Compute Unified Device Architecture.

OpenCL Open Computing Language.

ALU arithmetic logic unit.

DRAM Dynamic Random Access Memory.

SM Streaming Multi-processor.

CU Compute Unit.

COO Coordinate Format.

CSR/C Compressed Sparse Row/Column.

BCM Block Circulant Matrix.

FFT Fast Fourier Transform.

IFFT Inverse Fast Fourier Transform.

Acknowledgments

I want to thank my advisor Prof. David Kaeli, who has selflessly shared his knowledge and experience with me. I could not imagine completing my Ph.D. dissertation without his great support and guidance. His advising not only impacts my career in Computer Engineering but also my life. He showed me important traits to be a successful mentor, i.e., work ethic, leadership, and, most importantly, compassion. I feel blessed to be his student.

I want to thank many great professors, including Prof. Jennifer Dy, Prof. Rafael Ubal, Prof. José L. Abellán, Prof. José Cano, and Prof. Xue Lin. I sincerely appreciate their invaluable advice and feedback for completing this dissertation.

I want to mention my previous/current colleagues and friends of NUCAR. I am privileged to work, play, and celebrate with such a talented group of people.

I want to thank my best friend Xijun, my parents and my brother for their unreserved support and encouragement.

Last but not least, I want to thank my wife, Xin Gu, for her love, tolerance, and always being there.

Thank SBTS2018 from flaticon.com for providing icons for the diagrams of this dissertation.

Abstract of the Dissertation

Exploring High Performance Deep Neural Networks on GPUs

by

Shi Dong

Doctor of Philosophy in Electrical and Computer Engineering

Northeastern University, August 2020

Dr. David Kaeli, Advisor

Over the past few decades, Machine Learning (ML) has gained unprecedented popularity, becoming a pervasive technology that has benefitted a broad range of domains such as market analysis, environmental science, medical research, and material science. Among the many ML algorithms, Deep Learning (DL) has been shown to be effective in terms of accuracy, becoming the primary focus of attention for many leading research and industrial labs. The primary algorithm used in DL is a Deep Neural Network (DNN), a deep-structured multilayer Artificial Neural Network (ANN). To achieve better accuracy, researchers keep adding more layers and parameters when building a DNN model. As a result, new computing challenges emerge as a larger DNN model places more demands on computing and memory resources, especially during the training phase. Given their massive degree of parallelism, GPUs are a good fit for accelerating DNN training. However, DNN execution on a GPU can be highly inefficient, limiting their potential to accelerate deeper DNN models.

We need to develop more efficient hardware acceleration in order to enable future scalability of DNNs. There are three fundamental challenges faced when running high performance DNNs on a GPU. First, there is a void of tools available to measure and tune the performance of DNN computations on a GPU. Second, there is limited prior work with regards to characterizing bottlenecks present in this class of workloads when targeting GPUs. Lastly, existing methods that utilize sparsity and model compression have targeted FPGAs and ASICs, but not on GPUs. DNNs run on GPUs exhibit unique execution patterns. Directly applying prior techniques developed for FPGAs and ASICs can lead to significant inefficiencies, potentially degrading performance.

In this dissertation, we develop DNNMark, a GPU benchmark suite that consists of a collection of DNN primitives, covering a rich set of GPU computing patterns. This suite is designed to be a highly-configurable, extensible, and flexible framework in which benchmarks can run either individually or collectively. Next, we characterize the performance bottlenecks present in a

Convolution Neural Network models by considering microarchitectural-level bottlenecks on a layer-by-layer basis. We also characterize the memory access behavior in the context of a typical GPU memory hierarchy. Furthermore, we present the design of Spartan, a lightweight hardware/software framework to accelerate DNN training on a GPU. Spartan provides a cost-effective and programmer-transparent microarchitectural solution to exploit the sparsity detected during training. Spartan consists of three components: i) a sparsity monitor that intelligently acquires and tracks activation sparsity with negligible overhead; ii) a tile-based sparse GEMM algorithm that leverages a new sparse representation, namely ELLPACK- DIB; and iii) a novel compaction engine designed specifically for GPU workloads to support dynamic compaction of sparse data into the ELLPACK-DIB format. Last, we explore acceleration of DNNs using Block Circulant Matrices (BCMs), a model compression technique. We identify the GPU-specific challenges posed by using BCMs. Then, we perform both general and GPU-specific optimizations that impact: i) the decomposition and interaction of individual operations required in BCM algorithm, and ii) the overall GPU kernel design.

Chapter 1

Introduction

Over the past few decades, Machine Learning (ML) has gained unprecedented popularity, becoming a pervasive technology that benefits a broad range of domains such as engineering, environmental science, medical research, and biology [1]. ML is a cognitive learning method, aimed at automatically learning a mathematical model from experience, i.e., training data. The learned model performs tasks such as classification, regression, feature selection, and clustering [2, 3]. ML emerges as a general data-driven approach to drive both academic and industrial domains. For example, ML has been utilized to develop efficient and effective software packages [4], identify environmental factors tied to preterm birth [5], annotate gene sequences [6] and predict the probability that someone will have cancer [7]. Massive amounts of data have been collected through the advances in data acquistion technologies, and storage devices, providing us with an unlimited source of information for training ML algorithms and facilitating the development of ML.

The abundant availability of data has spurred the development of ML solutions. A few decades ago, traditional mainstream media such as newspapers, TV and radio dominated the creation and dissemination of information/data. However, the data generated have many limitations for ML algorithms due to the amount and format of data. Today, the pattern of information/data creation and dissemination has drastically evolved thanks to the rapid development of telecommunication technologies for both backbone infrastructures and terminal devices. New forms of media carrying and information transmission have emerged. Gathering and creating information have become ubiquitous with virtually no limitation. Besides, digitized data has become the primary format for storing all types of information, allowing computers to access the data more readily. One study has shown that, by 2020, the amount of digital data processed worldwide could reach 35 trillion gigabytes per day [8]. Large scale datasets (Big Data) can provide a massive amount of information

sources for researchers to explore the potential of ML, leading to new discoveries.

Generally, ML consists of two significant steps, training and inference. primary goal of training is to learn a statistical/mathematical model from a training dataset. An inference is made by running the trained model while performing a specific task. Usually, the mathematical model can be represented by: i) a series of parameters W, and ii) a hypothetical rule H represented by a set of formulas/functions/algorithms incorporating the parameters. Once trained, the model outputs a result using the inputs X and the parameters W, based on a set of rules. The result can then represent a prediction, i.e., a category in classification problems or a value in regression problems.

For training an ML algorithm, we should formalize an objective function targeting minimizing the error or loss defined based on the hypothetical rule. Depending on whether or not the training process includes the ground truth (labels), the ML has two categories, i) supervised and ii) unsupervised learning. Equation 1.1 and 1.2 present the general formulations of objection functions for supervised and unsupervised learning, respectively.

$$\hat{W} = \underset{W \in \mathbb{R}}{\operatorname{argmin}} \, \mathbb{L}(H(X); O) \tag{1.1}$$

$$\hat{W} = \underset{W \in \mathbb{R}}{\operatorname{argmin}} \, \mathbb{L}(H(X)) \tag{1.2}$$

, in which $\hat{W}$ is the set of estimated parameters, $\mathbb{L}$ is the loss function, H is the hypothetical assumptions, X is the input and O is the ground truth/labels. Depending on the availability of O, the loss function $\mathbb{L}$ takes on different forms accordingly.

As observed in the equations, the supervised learning incorporates outcomes/labels in establishing the objective function, whereas, unsupervised learning only considers the hypothetical assumptions. A straightforward example of supervised learning is Linear Regression (LR) [9]. LR uses a linear transformation as the hypothesis (i.e., $Y = W \times X + b$, in which W is a vector of coefficients and X is the inputs, b is the bias, and Y is the outputs). The hypothesis is used to interpret the relationship between the independent variables (X) and dependent variables (Y). Since LR is a supervised learning method, the training data consists of the ground truth O associated with the inputs X. The LR learns the W so that the value of outputs Y is close to the O. The commonly used objective function for LR is the one that minimizes the mean squared error between Y and O [9]. Unsupervised learning, on the other hand, does not depend upon a ground truth. Therefore, the unsupervised ML algorithms mainly leverage the inherent characteristics of the input data to achieve the goal set by the objective functions. For example, one popular clustering algorithm, KMeans,

uses Euclidean distance to group data into K clusters [10]; one feature reduction algorithm, PCA (Principal Component Analysis), uses the covariance matrix of input data to construct the principal orthogonal components to produced reduced dimensionality [11].

Among supervised learning algorithms, one specific algorithm, Deep Learning (DL), has shown prominent performance in terms of accuracy, becoming the primary focus of attention for many leading research and industrial labs. The primary algorithm of DL is a Deep Neural Network (DNN), a deep-structured multilayer artificial neural network. Similar to other ML algorithms, DL is also data-driven. The training of a DNN requires a tremendous amount of training data. The primary goal is to learn a set of weights associated with a DNN from the training data. The trained DNNs can approximate any continuous discriminant function. Unlike other ML algorithms, DL can perform tasks in a more generalized manner on data in its raw format. For example, most conventional image processing algorithms first extract image features by applying domain-specific techniques targeting color, texture, and shape [12, 13, 14], then classify them using an ML algorithm such as Nearest Neighbor. The DNNs, on the other hand, are trained to process raw data in a manner that general knowledge of the features is automatically stored. As a result, for image recognition problems, DNNs are capable of recognizing thousands of image classes [15] with high accuracy [16, 17, 18, 19]. In some fields, such as image and speech recognition, DL can even outperform a human being [20]. When appropriately trained, DL can perform many different tasks such as image classification, object recognition/tracking, natural language processing, speech recognition, and artificial intelligence, with high accuracy. These tasks, in turn, have inspired many great products, reshaping our society and changing our daily lives.

Depending on the type of inputs, emerging produce markets have leveraged DL across multiple areas including but not limited to: internet of things, financial services, health care, advertising, and transportation. Figure 1.1 shows a selected set of emerging products leveraging DL, based on the type of inputs, i.e., image/video, speech, and text. For products with image/video and speech as inputs, the Internet of Things (IoT) devices use DL to recognize faces or voices, making accurate responses through human-to-computer interfaces [21]; the autonomous vehicles use DL to analyze the road condition and surrounding environments, automatically steering the wheel without the presence of a driver [22]; the medical devices, e.g., CT, MRI scanner and electrical microscopy, use DL for health care tasks such as medical image segmentation, lesion detection and image reconstruction across many sub-categories of neuro, retinal, pulmonary, digital pathology, breast, cardiac, abdominal and musculoskeletal, significantly outperforming traditional methods [23]; the virtual assistants, e.g., Siri from Apple and Alexa from Amazon, use DL to process speech

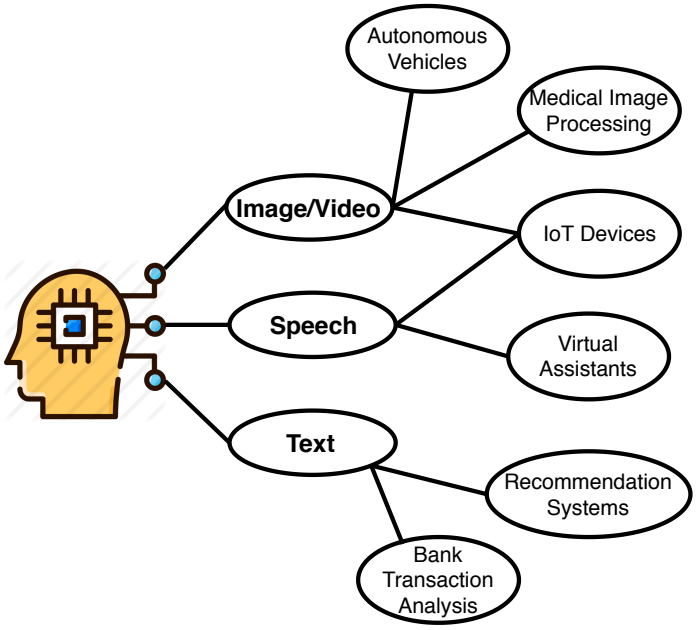


Figure 1.1: Products of Deep Learning based on type of inputs.

for obtaining lexical contexts, taking corresponding actions [24]. For products with text as input, banking/financial systems use DL to analyze bank transactions, detecting potential financial fraud and securing transactions [25]; advertising platforms use DL to produce user profiles and detect preferences, developing advanced recommendation systems, accurately sending advertisements to targeted customers [26].

1.1 Deep Neural Networks

The Deep Neural Network (DNN) plays an essential role in DL algorithms. The primary goal of a DNN is to systematically learn a set of weights, transforming the input data into representations/features and performing different tasks (e.g., image classification) based on the representations. The fundamental building block of a DNN is an artificial neural network (ANN). The essential element of ANN is an artificial neural/perceptron, a simple mathematical model simulating the process of a brain neuron [27]. Figure 1.2a illustrates the perceptron model. As shown in the figure, the perceptron consists of many weighted edges (W) connecting the input data X, a linear transformation involving operations such as multiplications and summations, and a non-linear activation function. Equation 1.3 details the computations in the model. First, a linear transformation involving only

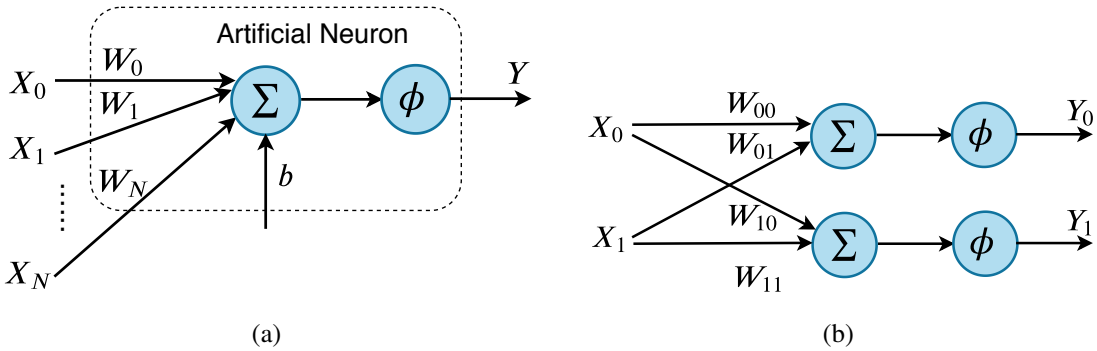


Figure 1.2: (a) Model of an artificial neuron. (b) A simple example of an ANN.

multiplications and summations outputs a set of pre-activations. Then, the non-linear activation functions ϕ output the final activations Y, performing a non-linear transformation. We will discuss the details in chapter 2.

$$\mathrm{Y} = \phi\Big(\sum_{i=0}^{N} \mathrm{W_i X_i} + \mathrm{b}\Big) \tag{1.3}$$

An ANN is composed of multiple perceptrons, with the weighted edges fully or partially connecting the input data. Figure 1.2b shows a simple example of an ANN (fully-connected) involving two perceptrons. To achieve high accuracy, a DNN adopts a multilayer design with a deep structure, in which each layer represents a type of ANN with a set of learnable weights/parameters. Conventionally, a DNN consists of one input layer, one output layer, and many hidden layers. Each layer represents an ANN of one specific type. Many other special-purpose layers may be incorporated to achieve better training performance and accuracy (Section 2). Figure 1.3 demonstrates one example of a feedforward DNN with two hidden layers for image classification. The input layer is responsible for transforming the input data into representations/feature maps for the following hidden layers. Note that for other problems, e.g., speech recognition and text mining, the input layer is designed to take speech/voice or text data as inputs. The hidden layers perform a series of transformations, extracting meaningful features/representations which become the inputs of the output layer. Different tasks can lead to different designs of the output layer. For example, the output layer for image classification problems generates a list of scores, each of which corresponds to a specific image class. As shown in Figure 1.3, the class with the highest score will be selected as the final prediction.

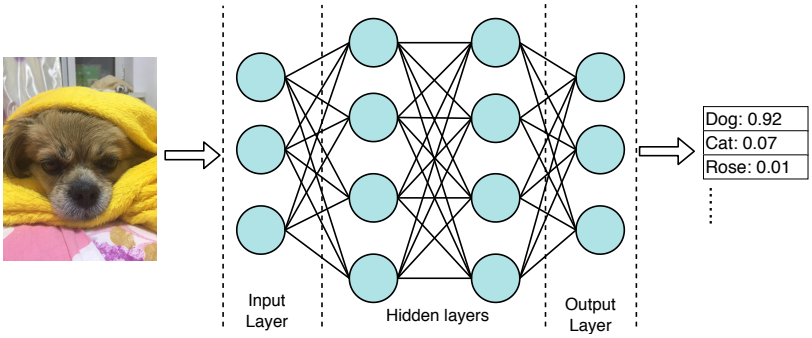


Figure 1.3: An image classification example using a feed-forward DNN.

1.1.1 A Brief History of DNN

The development of DNNs closely correlates to the development of Artificial Intelligence (AI). The foundation of DNNs, the artificial neuron, is one of the early inventions among many rudimentary AI technologies. Inspired by actual brain activities, McCulloch et al. published one work in 1943, exploring mathematical models simulating biological neurons [28]. One decade later, Rosenblatt proposed the artificial neuron/perceptron and neural network [27], sparking the first AI trend, together with other AI inventions such as game AI [29] and the Logic Theorist [30]. Although it looked quite promising, the neural network then was highly criticized for its limitations of addressing real and complicated problems, eventually leading to the end of connectionism (the ideology of connecting multiple artificial neurons to build a neural network) [31]. In the early 1980s, multiple independent research work had brought the neural network back to public attention. Hopfield proposed a new neural network model, i.e., Hopfield net [32], which led to the development of recurrent neural networks (RNN) [31]. Rumelhart et al. proposed the backward propagation, a method to propagate errors while training neural networks [33]. The backward propagation later enables the design of multilayer perceptrons (MLP). This became the fundamental design used by LeCun et al. (LeNet) to recognize handwritten digits, demonstrating the enormous potential of DNNs in the field of image recognition, using image 2D convolution [34].

However, for a very long time, the development of DNNs was slow due to the lack of: 1) training data and 2) required computational power. Thanks to the advancement of Big Data related technologies and advances in computing platforms (i.e., multi-threaded and parallel computing platforms), a new era of AI was born, creating the third wave of AI dominated by the DNN [35]. In the late 2000s, Li et al. launched the ImageNet project, aimed at providing an image database targeting image classification problems, facilitating the research in the DL field [15]. To date,

the ImageNet database has more than 14 million hand-annotated images corresponding to 21,841 categories. This number keeps growing at a fast pace as more and more image data becomes available each day. Besides, the emergence of high performance parallel computing platforms such as Graphics Processing Units (GPUs) has further facilitated the processing of DNNs, enabling large scale deployments of DNNs possible. With the computing resources of cutting-edge platforms, we can significantly shorten the processing time of large scale DNNs. As DNN models grow in terms of depth (layers) and width (in-network parameters), the full potential of a DNN will continue to grow.

1.1.2 Variants of DNNs

The architectures of the DNN model have evolved dramatically, resulting in multiple variants for different kinds of problems. Based on the neuron connectivity pattern, a DNN has two major categories: i) a Feedforward Neural Network, and ii) a Recurrent Neural Network (RNN). The feedforward neural network adopts a straightforward structure, as shown in Figure 1.3. The neuron connection in the feedforward neural network is acyclic, whereas the RNN has a cyclic connection pattern, in which the output of one layer is also the input of the same layer. The connection pattern of a RNN is aimed at capturing the temporal context of the input, such as speech or text.

A Convolutional Neural Network (CNN) is one class of feedforward neural network when 2D image convolution serves as the linear transformation layer. The 2D image convolution has a long history in image processing, mainly used to perform image transformations or feature extraction [36]. When used in a DNN, the convolution is a partially connected neural network, with connectivity patterns defined by the image convolution. A convolution layer is highly useful in a CNN for processing image data [34]. Almost all popular DNN models for image classification are CNNs [16, 18, 17, 19].

1.2 Platforms of DNN Execution

Different phases of DNN execution require different types of computing platforms. As a result, there are two major computing platforms commonly used for DNN execution: 1) cloud computing and 2) edge computing. The cloud computing is a server-based platform, providing tremendous amounts of storage and computing resources on demand. Usually, a cloud computing platform is designed to be a distributed system with many high-end computing and storage nodes. When handling the execution of a DNN, cloud computing platforms target the most resource-

demanding phase of DNN, i.e., training. As mentioned above, the training of a DNN requires a significant amount of training data in order to have a robust and reliable trained model. The training process of a DNN produces a heavy computational demand, comprising a forward and backward propagation (details will be described in chapter 2). Both the computing and storage resources present in traditional computing platforms (e.g., a desktop computer) are insufficient. As a result, cloud computing platforms become a good fit for running the training phase of using a DNN. Edge computing platforms (e.g., a cellphone), on the other hand, have limited computing resources, as well as power constraints. Therefore, the inference phase of using a DNN is the primary focus of edge computing platforms. Specifically, DNN inference only performs computations for the forward propagation, with only a handful of inputs. Even though having many limitations and constraints, edge computing platforms still have significant advantages over cloud computing platform in some use cases, eliminating the communication overhead between the cloud (server) and edge devices (users).

For different types of computing platforms, a variety of available hardware is available for running a DNN. The DNN can be viewed as a scientific computing problem, where we can use CPUs, GPUs, FPGAs or ASICs as the primary hardware platform for acceleration. According to the fundamental mathematics of artificial neurons, matrix-based operations, such as element-wise operations, matrix transformations and multiplications, dominate DNN computations. As such, hardware platforms that can accelerate matrix-based operations are targeted to run DNNs. Matrix-based operations are inherently parallel. For example, the vector-add operation can be done by adding every element of the two input vectors entirely in parallel, reducing the timing complexity from $O(N)$ to $O(1)$. On a multi-core CPU, simultaneous multi-threading [37] and vector instruction extensions such as Advanced Vector Extension (AVX)[38] are two techniques aimed at exploiting parallelism. A GPU, on the other hand, is designed initially to support a high degree of parallelism through architectures with thousands of cores, enabling thousands of threads running simultaneously. FPGAs and ASICs are customized architectural designs used to maximize parallelism according to the specific problem. Considering that FPGAs and ASICs allow customized architectural design, energy efficiency can be tuned and optimized. Cloud computing platforms used for DNN training, are commonly equipped with server-grade CPUs and GPUs, whereas edge computing platforms are targeting DNN inference, and are thus equipped with low-end CPUs and GPUs or customized accelerators leveraging FPGAs or ASICs. As an emerging platform, the TPU (Tensor Processing Unit) has become a popular platform for both DNN training and inference [39]. TPUs serve as a GEMM or tensor processing engine, controlled by a host. Therefore, we also consider it as a

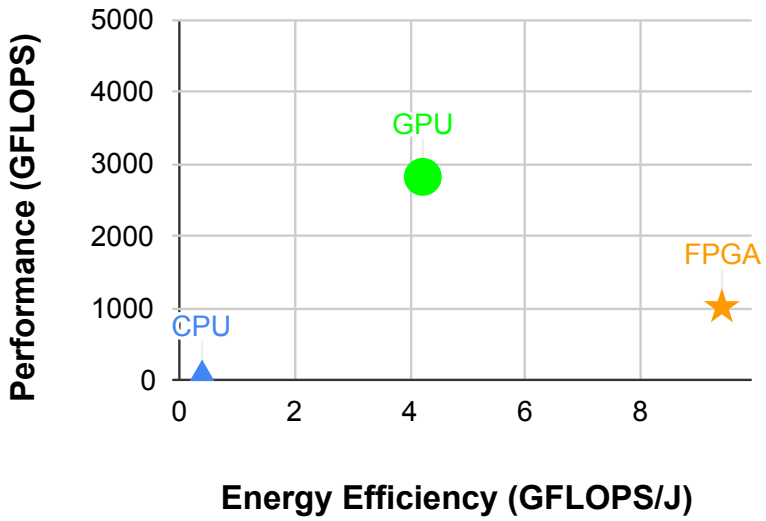


Figure 1.4: Performance vs. Energy efficiency of CPUs, GPUs and FPGAs when used for DNN training.

customized accelerator in this dissertation.

Figure 1.4 illustrates the differences between a CPU, a GPU, and a FPGA in terms of performance and energy efficiency during DNN training. We collect the corresponding data shown in the figure from [40, 41, 42]. We exclude ASICs in this discussion because ASICs are more commonly used for inference on resource-constraint devices. Accelerating DNN training will be the primary focus in this dissertation. From the figure, we can see that a GPU achives the best performance in GFLOPS. A FPGA provides for the best energy efficiency. In terms of either performance or energy efficiency, it is evident that the CPU is not a good target for running a DNN. In this dissertation, we target addressing the challenges when executing DNN training on a GPU.

1.3 Challenges of DNN Execution on GPUs

New challenges in computing emerge as researchers keep increasing the depth and width of the network. When adding more layers and parameters, the execution of a DNN places more computational demands on the hardware platform, resulting in training runs that can take days or even weeks to finish. Therefore, developing fast and efficient hardware for running DNNs becomes more and more critical to further shorten the training time, especially as DNN model continue to

grow in complexity. In this section, we discuss focus our discussion on the challenges of DNN execution on GPUs.

GPUs behave as accelerators in heterogeneous systems that incorporate both CPU and GPU. Generally, the main program is running on a CPU, while offloading the compute-intensive and parallelable kernels to a GPU to accelerate execution. As discussed in previous sections, the dominant computations in a DNN are matrix-based operations, which are both compute-intensive and parallelizable As such, GPUs are popular hardware platforms to run DNNs. It is important to investigate methods for fast and efficient DNN execution on GPUs. Below, we present three significant challenges facing high-performance DNN execution on GPUs.

First, there is a void of tools available to measure and tune the performance of DNN computations on a GPU. Prior work paid little attention to tuning the software performance. Existing DL frameworks cannot satisfy the needs of designers of next-generation hardware platforms, who desperately need a framework where they can evaluate design tradeoffs when executing DL algorithms. While some researchers have considered extracting DNN primitives and embedding them in reconfigurable hardware, this approach is too labor-intensive, and a customized solution will compromise the configurability.

Second, there is limited prior work with regards to the execution of this type of workload on GPUs. Given this limited knowledge, it becomes challenging to optimize GPU architectures to run this class of compute-intensive applications. A detailed characterization is required to understand the execution behavior of DNN execution on GPUs.

Lastly, existing methods have many limitations when run on GPUs. Optimizing DNN execution on GPU requires efforts on multiple levels, i.e., algorithm, software, and hardware. Existing techniques, such as utilizing sparsity and model compression, have been extensively investigated to improve the execution efficiency of DNNs. These have been targeted mainly to speed up customized hardware leveraging FPGAs or ASICs [43, 44, 45, 46]. However, the same techniques present completely different execution behaviors on GPUs. Directly applying those techniques can lead to significant inefficiencies, potentially degrading performance.

1.4 Contributions of This Dissertation

In this dissertation, we focus on investigating methods for high-performance DNN training on GPUs. Below we summarize the envisioned contributions of this dissertation.

- We develop *DNNMark*, a GPU benchmark suite that consists of a collection of DNN primitives, covering a rich set of GPU computing patterns. This suite is designed to be a highly configurable, extensible, and flexible framework in which benchmarks can run either individually or collectively. The goal is to provide hardware and software developers with a set of kernels that can be used to develop increasingly complex workload scenarios.

- We characterize performance implications of a CNN model using microarchitectural details on a layer-by-layer basis, and characterize the memory access behavior in the context of a typical GPU memory hierarchy, considering hardware resource utilization associated with each primitive in the CNN model. We identify the main bottlenecks by considering the potential limits of using a single GPU.

- We develop *Spartan*, a lightweight hardware/software framework, to accelerate DNN training on a GPU. More specifically, Spartan provides a cost-effective and programmer-transparent microarchitectural solution to exploit the sparsity detected during training. Spartan consists of three components: i) a sparsity monitor that intelligently acquires and tracks activation sparsity with negligible overhead; ii) a tile-based sparse GEMM algorithm that leverages a new sparse representation, namely ELLPACK-DIB; and iii) a novel compaction engine designed specifically for GPU workloads to support dynamic compaction of sparse data into the ELLPACK-DIB format.

- We optimize the General Matrix Multiplication (GEMM) using Block Circulant Matrices, a model compression technique, for DNN training on a state-of-the-art GPU. First, we identify GPU-specific challenges posed by using the structured matrix. Next, we perform both general and GPU-specific optimizations that impact: i) the decomposition and interaction of operations, and ii) the overall kernel design.

1.5 Organization of This Dissertation

We arrange the remainder of this dissertation as follows. Chapter 2 presents background knowledge, including: i) operations involved in each DNN primitive, ii) the training process of DNN, iii) the GPU architecture, and iv) the GPU programming model. In Chapter 3, we review related work that explores DNN acceleration from multiple perspectives, including benchmarking, characterization, optimization, and hardware design. In Chapter 4, we introduce the *DNNMark*

benchmark suite, and discuss its design and usage. In Chapter 5, we provide a CNN characterization, discussing observations about CNN behavior. In Chapter 6, we propose *Spartan*, presenting a novel design leveraging activation sparsity for accelerating DNN training. In Chapter 7, we present the work that explores GPU acceleration for DNN training using structured matrices. In Chapter 8, we conclude this dissertation and present future work plans.

Chapter 2

Background

In this chapter, we present the background material specific to this dissertation. First, we describe Deep Neural Network (DNN) primitives that are common in popular DNN models (i.e., AlexNet, VGGNet, GoogleNet, and ResNet), as well as the computational patterns of these primitives. We then describe the training process for a DNN, discussing the algorithmic steps required for training a DNN. Next, we review general GPU architecture, highlighting different GPU computing resources and the GPU memory hierarchy. We want to be able to launch thousands of threads concurrently and provide high bandwidth for memory transfers for data movement. We also present the GPU programming model, which allows researchers to utilize a GPU as a general purpose platform for accelerating a wide range of applications.

2.1 DNN Primitives

2.1.1 The Convolutional Layer

Convolution in a DNN involves a 2-D image convolution [47], an effective method for extracting features through multiple convolutional kernels/filters. Figure 2.1 shows an example of the 2-D image convolution process. In this figure we apply a 3×3 convolutional kernel to an area of the image with the same dimensions. We denote the input data as X, and the kernel weights as W. Using the computation shown in equation 2.1, we can calculate a single result for Y based on the data and the kernel as follows:

$$\mathrm{Y_{i,j}} = \sum_{h=0}^{k_h-1} \sum_{w=0}^{k_w-1} \mathrm{W_{h,w}} * \mathrm{X_{i+h,j+w}} \tag{2.1}$$

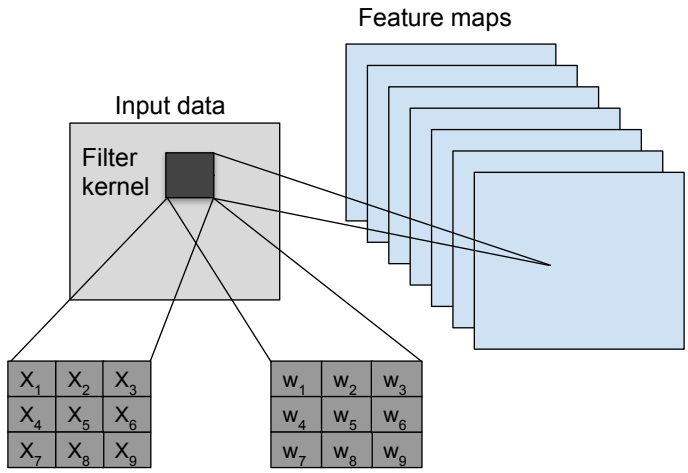


Figure 2.1: 2-D image convolution

,

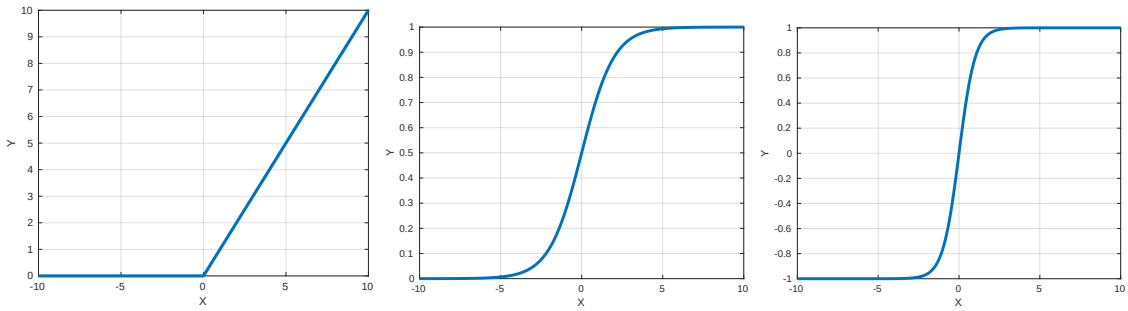
in which, the k_h and k_w are the height and weight of the kernel, i and j are the indices of the output feature map, and are used to specify the location of the input data sample.

To calculate an entire feature map, we move the kernel across every data sample and apply the same computation repeatedly. When moving the kernel across data samples, we can use a specified stride. As the stride grows, the calculated feature map has a narrower height and width. Usually, a convolution has multiple kernels to generate different feature maps, extracting a broad range of features.

Multiple algorithmic options are available to perform image convolution, such as Direct, GEMM, Fast Fourier Transform (FFT) [48] and Winograd [49]. The Direct algorithm expresses the convolution as a direct convolution as described in Figure2.1; The GEMM method transforms the entire process into a matrix-matrix multiplication; The FFT and Winograd algorithms have reduced timing complexity [49]. The former takes advantage of the mathematical properties of FFTs/IFFTs. The latter simplifies the computation needed to compute the convolution using kernels with a specific size [49].

2.1.2 Fully-Connected Layer

The fully-connected (FC) layer is an Artificial Neural Network (ANN) described in Chapter 1.1. The computation of the FC layer can be directly represented by matrix-based operations. For example, as shown in Figure 1.2b, we have one fully-connected layer with only two artificial



(a) (b) (c)

Figure 2.2: (a) ReLU. (b) Sigmoid. (c) Tanh.

neurons. Given this model, the output of this layer can be represented as:

$$\begin{aligned} Y_0 &= W_{00}X_0 + W_{01}X_1 \\ Y_1 &= W_{10}X_0 + W_{11}X_1 \end{aligned} \tag{2.2}$$

We can transform Equation 2.2 into matrix-vector multiplications as follows: i

$$\begin{bmatrix} Y_0 \\ Y_1 \end{bmatrix} = \begin{bmatrix} W_{00} & W_{01} \\ W_{10} & W_{11} \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \end{bmatrix} \tag{2.3}$$

When performing this set of operations across multiple sets of data inputs, the computation becomes a matrix-matrix multiplication (GEMM).

2.1.3 Non-linear Activation Function

The activation function is the non-linear step used in a DNN model for handling linear inseparable problems. Commonly used activation functions include: a Rectified Linear Unit (ReLU) (equation 2.4), a Sigmoid (equation 2.5), and a Hyperbolic Tangent (Tanh) (equation 2.6). The equations below provide mathematical descriptions of these functions.

$$\mathrm{y_i} = \max(0, x_i) \tag{2.4}$$

$$\mathrm{y_i} = \frac{1}{1 + e^{-x_i}} \tag{2.5}$$

$$\mathrm{y_i} = \frac{e^{x_i} - e^{-x_i}}{e^{x_i} + e^{-x_i}} \tag{2.6}$$

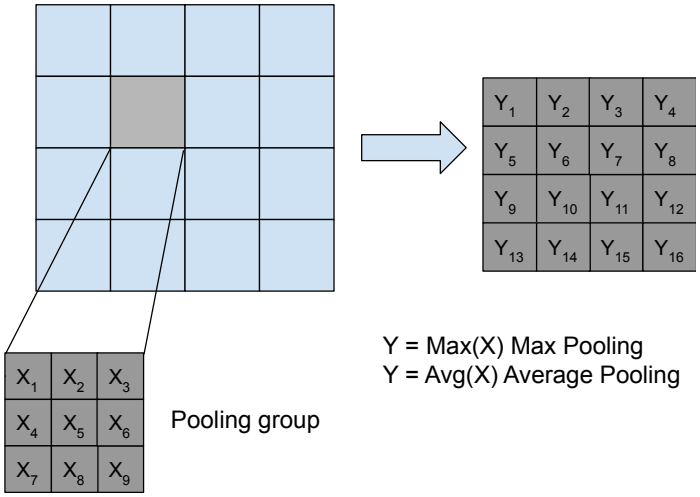


Figure 2.3: Feature map data divided into multiple pooling groups.

Considering non-linearity is critical in order to support the multi-layer structure of DNN. Convolutional and the fully-connected layers are essentially linear data transformations. Concatenating these two types of layers is equivalent to approximating a simple linear classifier, limiting a model's potential to address complex real-world problems [50]. However, adding a non-linear activation function between linear transformation layers allows the training to separate different linear transformation layers, making the multi-layer structure possible when approximating complex discriminant functions. Figure 2.2a- 2.2c plots the three activation functions, presenting their mathematical properties. ReLU is the most commonly used activation function due to its simplicity.

2.1.4 Pooling

Pooling operations play an essential part in a DNN model. The main goal of Pooling is to down-sample, thus reducing the amount of computation. Pooling is also a practical approach in practice to improve robustness, selecting the most representative value from a sub-group of the feature map. By doing so, we can significantly reduce the interference of neighboring pixels.

Figure 2.3 shows an example of the pooling process. First, we divide the input data into multiple pooling groups. In the figure we have 16 pooling groups, each of which contains 9 elements. The pooling groups are k data samples apart from each other, as shown in the figure. Here, k is the pooling group size (k is 3 in the example). Next, for each pooling group, we compute one result according to a specific pooling scheme (e.g., Max Pooling or Average Pooling). Max Pooling selects the maximum value in a pooling group. Average Pooling computes the average of the group.

2.1.5 Local Response Normalization

Local Response Normalization (LRN) is one technique inspired by a biological activity named *lateral inhibition* [51]. When applying LRN, the prediction error rate can drop by 2% [16]. LRN normalizes the input feature maps using equation 2.7 below. As described in the equation, to normalize x_i, LRN uses multiple x_j's from the adjacent feature maps, but at the same relative position of the corresponding x_i's.

$$\mathrm{y_i} = \frac{x_i}{k + \alpha(\sum_{j=\max(0,i-n/2)}^{\min(N-1,i+n/2)} (\mathrm{x_j})^2)^\beta} \tag{2.7}$$

where k, α, β are the configurable parameters of the LRN, N is the number of kernel maps, n is the window size for normalization, x_i is the input data, and y_i is the output with the same spatial location.

2.1.6 Batch Normalization

Batch normalization (BN) is one technique introduced to reduce the training time [52]. BN normalizes the distribution of a batch of input data before the non-linear activation function, leading to a data distribution with zero mean and unit variance. By varying the distribution of the input data we can significantly re-adjustment the learnable parameters during training [52]. Consequently, using standard DNN training with a faster learning rate becomes very challenging in terms of convergence. When the distribution of the input data remains constant over different iterations, the training process more quickly converges. For input data x with m data samples, the process of batch normalization can be represented by the following equation:

$$\hat{x_i} = \frac{x_i - \mathrm{Mean}[x]}{\sqrt{\mathrm{Var}[x] + \varepsilon}} \tag{2.8}$$

,

where x_i is a single sample indexed by i, $\hat{x_i}$ represents the normalized data, $\mathrm{Mean}[x]$ is the mean value of x, and $\mathrm{Var}[x]$ is the variance of x. The ε is included to prevent division by 0. Equations 2.9 and 2.10 show the computation of the mean and variance.

$$\text{Mean}[x] = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{2.9}$$

$$\text{Var}[x] = \frac{1}{m}\sum_{i=1}^{m}(x_i - \text{Mean}[x])^2 \tag{2.10}$$

2.1.7 Softmax

Softmax [53] is a commonly-used function in the output layer. It interprets the output data from the previous hidden layer and generates a set of scores in the range of 0 to 1 (note, the sum of all scores equals 1). For each value that Softmax computes, it represents the extent to which the input data should be classified into one of the predefined classes. The Softmax function is shown in equation 2.11:

$$Y_i = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}} \tag{2.11}$$

where N is both the number of outputs from the previous layer and the number of classes.

2.2 Popular DNN models

In this section, we describe three popular DNN models, i.e., AlexNet [16], VGGNet [17], and ResNet [19]. We select these three models because: i) they serve as good examples of DNNs incorporating all the primitives described in section 2.1, ii) they represent major milestones in DNN evolution, and iii) they have been used in a large number of prior studies, especially in areas of computer architecture and accelerator research [54, 55, 56, 43, 44, 45, 57].

Krizhevsky et al. first introduced AlexNet in 2012. AlexNet is a multi-layer CNN model incorporating layers that include Convolution, the Fully-Connected layer, Max-Pooling, LRN, and Softmax. Table 2.1 describes the details in terms of the number of different layers in AlexNet.

AlexNet employs ReLU as the activation function. For the convolutional layer, AlexNet uses three different convolutional kernel sizes, i.e., 11×11, 7×7, and 3×3. AlexNet uses 96, 256, and 384 as the number of channels for different intermediate feature maps. For the fully-connected layer, the number of output nodes is 4096, except for the last layer, whose number of output nodes equals the number of predictions by the model. The top1 and top5 accuracy [16] of AlexNet trained with ImageNet data is 62.5% and 83%, respectively, in image classification. The design of AlexNet

Layer	Number
Convolution	5
Fully-Connected	3
Max-Pooling	3
LRN	2
Softmax	1

Table 2.1: The number of different layers in AlexNet.

Layer	Number (A)	Number (B)	Number (C)	Number (D)
Convolution	8	10	13	16
Fully-Connected		3		
Max-Pooling		5		
Softmax		1		

Table 2.2: The number of different layers in VGGNet.

presents a significant improvement in accuracy, showing great potential in terms of improving performance by adding more layers to a neural network model [16].

The Visual Geometry Group (Simonyan et al.) introduced VGGNet in 2014. Training with ImageNet data, VGGNet presents an improved top1 and top5 accuracy (75.2% and 92.5%, respectively) with a deeper structure and more parameters than those used in AlexNet. Similar to AlexNet, VGGNet is also a CNN, heavily utilizing convolutional layers. VGGNet has many variants including a different number of layers, determined mainly by the number of convolutional layers deployed. For convolutional layers, VGGNet uses two convolutional kernel sizes, i.e., 3×3 and 1×1 (only present in one variant), providing a number of options for the number of feature map channels, i.e., 64, 128, 256, and 512. Table 2.2 describes the details in terms of the number of different layers in VGGNet. In the table, the labels A, B, C, and D stand for models with 11, 13, 16, and 19 weight layers, respectively.

In 2015 He et al. proposed ResNet that includes novel building blocks and residual units, needed to overcome the *training degradation* problem when adding too many layers in the neural network. ResNet can achieve a very deep structure with over hundreds of layers, without experiencing training degradation. Training with ImageNet data, the top1 and top5 accuracy of ResNet is 78.57% and 94.29%. The following equation presents the details of a residual unit:

Layer	18-layer	34-layer	50-layer	101-layer	152-layer
Residual Unit (A)	8	16			
Residual Unit (B)			16	33	50
Convolution (7×7)			1		
Fully-Connected			1		
Max-Pooling			1		
Softmax			1		

Table 2.3: The number of different layers in ResNet.

$$Y = \phi\big(\mathbb{F}(X) + X\big) \tag{2.12}$$

, in which the $\mathbb{F}$ represents a series of linear transformations, combined with activation functions. ϕ is an additional activation function.

Similar to VGGNet, ResNet also has multiple variants, including a different number of residual units and different types of residual units as well. One type of residual unit (A) has two convolutional layers using 3×3 kernels, two batch normalizations, and one activation function in $\mathbb{F}$. Another type of residual unit (B) has three convolutional layers using 3×3 and 1×1 kernels, three batch normalization layers, and two activation functions. The number of channels in the feature maps in ResNet can be 64, 128, 256, 512, 1024, and 2048 channels. ResNet consists of one convolutional layer, multiple residual units, and one fully-connected layer. The convolutional layer serves as the first layer to transform input data using a 7×7 kernel. The fully-connected layer is the last layer to generate outputs before softmax. Table 2.3 describes the details in terms of the number of different layers in ResNet.

Figure 2.4 presents the number of layers and estimated GFLOPs needed for AlexNet, VGGNet, and ResNet. From the figure, we can find a clear trend that the DNNs are becoming deeper and deeper over time. The number of operations for computing the DNN is also significantly growing. Note that for VGGNet and ResNet we use the metrics from the deepest variant.

2.3 Training of DNN

Training a DNN is an iterative task. For each iteration, the training process updates the network parameters so that the network loss shrinks and eventually converges. One of the most

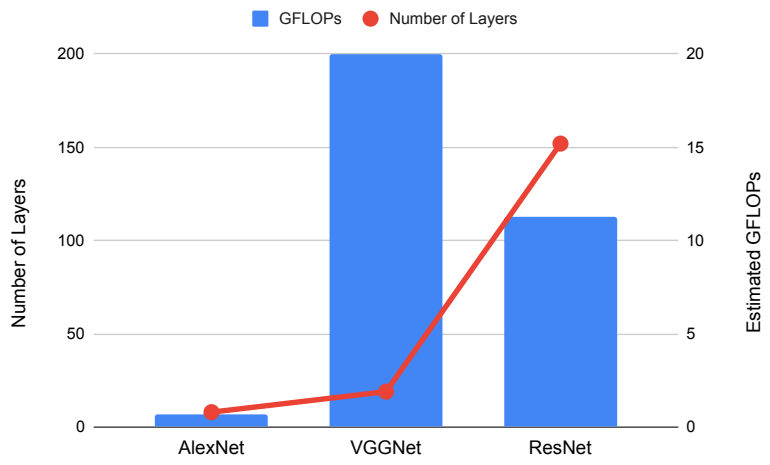


Figure 2.4: The number of layers and estimated GFLOPs needed for AlexNet, VGGNet, and ResNet.

effective algorithms used to update the parameters is *Stochastic Gradient Descent* (SGD), which processes a mini-batch of the training data on each iteration. SGD can be expressed in equation 2.13.

$$\mathrm{W_{i+1}} \leftarrow \mathrm{W_i} - \alpha \sum_{n=1}^{m} \frac{\partial L}{\partial W_i} \tag{2.13}$$

where m is the size of the data in a mini-batch, W_i are the parameters on iteration i, W_{i+1} are the updated parameters for iteration $i+1$, α is the learning rate, a hyber-parameter adjusting the speed of updating the network, and $\frac{\partial L}{\partial W_i}$ is the derivative of the loss function with respect to the parameters W_i.

The detailed mechanism for training consists of a forward and a backward propagation. The forward propagation performs layer-by-layer transformations of the input data based on the current parameters, calculating the network loss. The backward propagation calculates derivatives of the calculated loss with respect to the inputs $\frac{\partial L}{\partial X_i}$, outputs $\frac{\partial L}{\partial Y_i}$, and parameters $\frac{\partial L}{\partial W_i}$ of each layer. Similar to the layer-by-layer transformations performed during the forward propagation, the backward propagation applies the derivative chain rule for each layer in a backward-cascaded fashion. Next, the SGD algorithm takes $\frac{\partial L}{\partial W_i}$ as its inputs for updating the network parameters.

After training is complete, we can then compute the inference through another forward propagation, but this time we skip the calculation of the loss. The results of the inference are final *predictions*, which should be a vector containing probabilities for each label from the predefined classes. We can also scale the outcome to allow prediction accuracy to seem more intuitive.

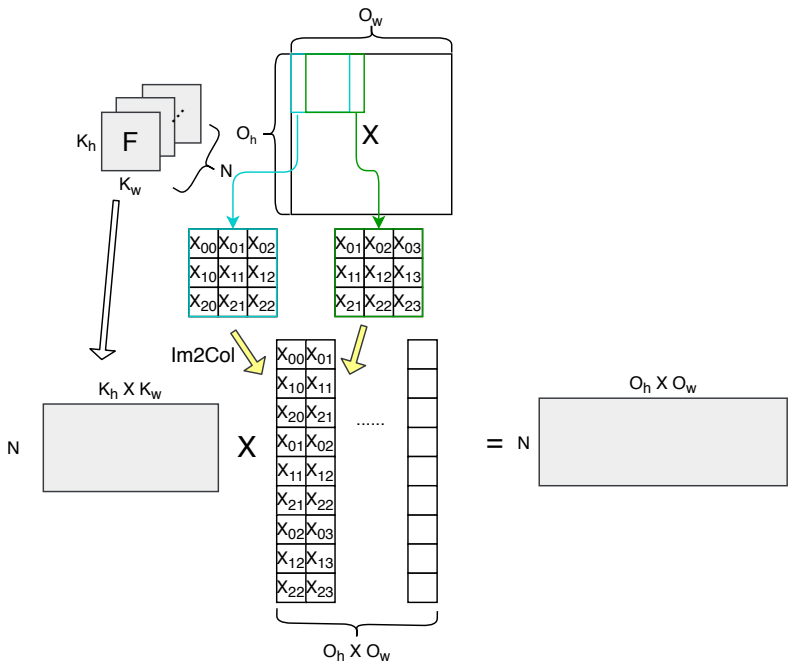


Figure 2.5: An example of transforming a convolution into GEMM operations using the im2col transformation.

In practice, the training process for a DNN can be very time-consuming. It usually requires many hundreds of thousands of iterations before the network parameters are adequately trained. As newer DNN models become deeper and have a larger number of parameters, each iteration can take longer to finish, exacerbating this situation.

2.3.1 DNN Computation

When training a DNN, matrix-based MAC operations (i.e., multiply-accumulates) dominate roughly 90% of the overall execution time [58]. The primary source of the matrix-based MAC operations occur in General Matrix Multiplication (GEMM) used by linear transformation layers (i.e., convolutional and fully-connected layers).

The convolutional layer can use GEMM as its computational method after applying an im2col operation to its inputs [59]. The im2col operation transforms the input into a matrix. After applying the transformation, the convolution operation described in Equation 2.1 becomes a GEMM operation. Figure 2.5 shows an example of transforming a convolution into a GEMM operation using im2col. In the figure, we have N filter kernels of size $K_h \times K_w$ (where $K_h = K_w = 3$)

and one feature map of size $O_h \times O_w$. To calculate the first output element, we perform a MAC operation with the kernel filter and a tile of the feature map (outlined in blue), which includes $X_{00}, X_{10}, X_{20}, X_{10}, X_{11}, X_{12}, X_{20}, X_{21}, X_{22}$. The im2col operation unrolls the tile into a column, as shown in the figure. We can see that the MAC operation becomes a dot product of one row of the weight matrix, corresponding to one kernel filter, and one column, corresponding to the tile of the feature map. Next, we perform the same process on the second tile (outlined in green) and, eventually, all the feature map tiles. Then, the convolution becomes a GEMM, involving one matrix of size $N \times K_h \times K_w$ and a second matrix of size $K_h \times K_w \times O_h \times O_w$. The output of the GEMM is a matrix with size $N \times O_h \times O_w$. The fully-connection layer is inherently a GEMM operation, as described in section 2.1.2.

2.3.1.1 Sparse GEMM

Sparse GEMM has been widely studied to accelerate high-performance computing applications. The key applications include sparse linear systems and graph algorithms [60]. Leveraging sparse GEMM reduces the number of computations by skipping multiplications involving zero. In addition, Sparse GEMM compresses a matrix with a large number of zeros, saving memory space needed for storage. Researchers have developed many different types of compressed sparse formats and associated sparse GEMM algorithms. The popular sparse formats to compress a sparse matrix include COO, CSR/C, HYB, and ELLPACK [61].

DNN computation can also benefit from using sparse GEMM for both weight and activation maps, since both can be sparse. The sparseness in the weight matrix can be introduced by weight pruning and quantization [62, 63]. The sparseness in the activation maps is usually a product of the ReLU or Max-Pooling layers [64]. Given their computing nature, ReLU and Max-Pooling can generate zeros in a DNN.

2.4 Graphic Processing Units (GPUs)

2.4.1 Architecture

Unlike a CPU architecture, the architecture of a Graphic Processing Unit (GPU) improves instruction throughput rather than reducing the latency of a single instruction. As such, the compute unit organization is much simpler than a CPU core design. That allows the GPU to include many more cores than a CPU. The recent trend suggests there will be more compute resources as famous GPU

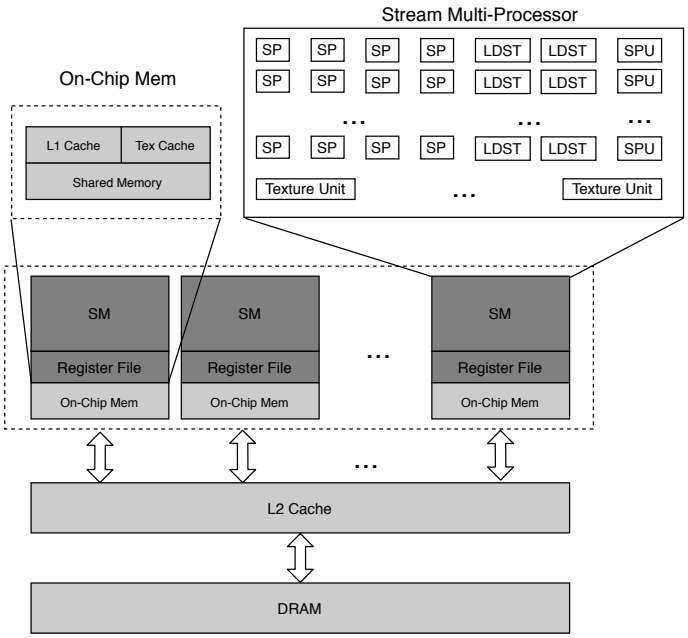


Figure 2.6: Diagram of a typical NVIDIA GPU architecture.

vendors are using smaller nanotechnology (i.e., 7nm) [65]. Currently there are thousands of cores available to run thousands of threads simultaneously. A GPU has support for low-overhead thread switching to hide latency. GPUs can outperform CPUs in most cases where instruction throughput matters. A good example is with matrix-based computations, which are easily parallelizable. GPUs are well-suited to execute matrix-based operations that are designed to run on parallel platforms.

The basic building block on an NVIDIA GPU is called a multi-threaded Streaming Multiprocessor (SM). On an AMD GPU, the same element is called a Compute Unit (CU). GPUs from both NVIDIA and AMD utilize a very similar architectural design. Thus, for simplicity, we will stick to the architecture and terminology associated with an NVIDIA GPU in this dissertation. We will mention the differences between an NVIDIA GPU and an AMD GPU when they occur. Figure 2.6 illustrates the architecture of an NVIDIA GPU. As shown, each SM contains a collection of computational resources that includes single precision cores, load-store units, special functional units and texture units. Each SM is also equipped with a large register file so that threads can have their own set. Providing dedicated registers for threads eliminates the need to swap out threads during context switching (as required on a CPU), which significantly reduces the corresponding overhead. With a low-overhead context switch, threads can hide pipeline stalls and effectively utilize the full computational resources of the GPU.

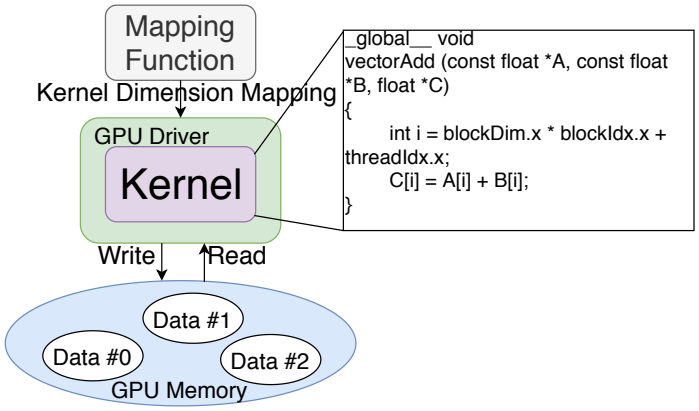


Figure 2.7: GPU kernel design process (a vector add kernel design).

An array of SMs is connected to a hierarchical memory system. Each SM has limited memory resources that are exclusive to each SM, including an L1 cache, a shared memory, and a read-only/texture cache. The shared memory is a scratchpad cache that is accessible by the programmer. Each SM has exclusive memory resources connected to a shared L2 cache. The L2 cache is the primary point of data unification between the SMs, servicing all general and texture memory requests, providing efficient, high-speed data sharing across the GPU. The L2 cache is backed by a high bandwidth DRAM memory, which is the largest memory on a GPU that programmers can access directly.

2.4.2 Programming Model

The programming model on a GPU provides an intuitive abstraction of hardware for programmers. When programming a GPU for general purpose, programmers use a run time library and follow a specifically defined programming model to operate the GPU. CUDA provides programmers a full-fledged toolset to run a program on NVIDIA GPUs; ROCM on AMD GPUs. The programmer designs a kernel function that can be executed by each thread on the GPU. Given the Single Program Multiple Data (SPMD) programming model on the GPU, threads are grouped into thread blocks, and thread blocks are grouped into a grid. Within each thread, a unique index can be obtained using the thread ID and block ID. The blocks and threads can be grouped in a 2D or 3D manner, meaning that the thread ID and block ID are 2D or 3D (a CUDA-specific identifier, *threadIdx* is used to obtain the thread ID, *blockIdx* for block ID) [66]. In practice, a thread block contains a number of warps (NVIDIA) or wavefronts (AMD), the basic unit containing a specific number of threads scheduled on the hardware (32 for NVIDIA GPU and 64 for AMD GPU).

Before launching the kernel, programmers should carefully design a mechanism to map the specific problem on to a number of threads with a specific structure (i.e., the kernel dimension) to support flexible data mapping and provide potential performance benefits. Figure 2.7 shows the essential elements required to design a GPU kernel: i) mapping the data layout to the kernel dimensions (i.e., the shape of the cooperative thread array, defined by blockDim and gridDim), ii) specifying the operations to be performed in the kernel, and iii) managing the kernel input data, stored in GPU memory while the kernel is active.

Chapter 3

Related Work

In this chapter, we review prior work related to this dissertation. DNN research can be categories into three broad classes: 1) algorithms, 2) applications and 3) platforms. In this dissertation, we target research literature focusing on DNN platforms, benchmarking, optimization and hardware designs for DNNs. We target a GPU as the primary hardware platform. However, given the popularity of customized hardware designs for DNNs on an FPGA and ASICs, we also include them in this chapter. We partition related work into three sections, reviewing related studies from three different categories: 1) DNN benchmarking, 2) DNN optimization, and 3) DNN hardware design.

3.1 Benchmarking and Deep Learning Frameworks

Benchmarking plays a critical role in the development of computing hardware. Researchers have widely used benchmarking techniques for two purposes: i) serving as performance indicators when presenting their novel architectural designs, and ii) investigating the execution behavior on specific platforms such as a GPU. In industry, manufacturers may use benchmarking to demonstrate the performance of their latest product or to compare multiple products.

In general, it requires a collection of applications to benchmark a computing platform thoroughly. The collection of applications is called a benchmark suite. Usually, applications included in a benchmark suite should meet the following criteria: i) the selected applications should have diverse workload characteristics, placing varied and sufficient demands on the compute and memory resources of a target platform, ii) the selected applications should serve a specific purpose. For example, HeteroMark [67] captures the computing patterns of CPU-GPU collaborative executions. DNNMark [68], proposed in this dissertation, is a benchmark suite explicitly designed to

benchmark DNNs on GPUs. Researchers run the same collection of applications on different hardware or architecture designs to observe performance differences while studying valuable execution implications.

There are many available benchmark suites for GPUs. For benchmarking general GPU performance, popular benchmark suites include Rodinia [69] and Parboil [70], providing a wide range of applications that include simulation, scientific computing, and machine learning on GPUs. HeteroMark [67] includes applications presenting three CPU-GPU collaborative execution patterns: i) workload partitioning, ii) producer-consumer, and iii) pipelining. Similar to HeteroMark, Chai [71] also provides a collection of collaborative heterogeneous applications, presenting two new patterns: i) coarse-grained and ii) fine-grained workload partitioning. NUPAR [72] provides GPU applications leveraging advanced GPU features, such as concurrent kernel execution and dynamic parallelism. Tartan [73] and MGMark [74] are two recent benchmark suites targeting multi-GPU environments. The former provides a set of customized programs to evaluate the performance of the interconnection of a multi-GPU environment. The latter enables the multi-GPU execution of HeteroMark. All of these benchmark suites have been used to study the performance of systems incorporating GPUs.

Deep Learning (DL) represents an important class of applications, so benchmarking DL becomes essential to achieve efficient execution future DL models. Some popular DL frameworks, such as Caffe[75], TensorFlow[76], and PyTorch[77], all provide some rudimentary functionality to benchmark DNNs for general execution performance. With support for toggling GPU usage, users can measure the performance of a specific DNN model on CPUs or GPUs. However, benchmarking is not the primary goal of these frameworks. They lack the ability to study the impact of the targeted microarchitecture in terms of performance, and lack sufficient configurability.

To bridge the gap, researchers have developed many benchmark suites, providing DNN benchmarking across different hardware platforms. Large enterprises from industry are spending significant efforts on this type of work. Baidu developed DeepBench [78], providing comprehensive benchmarking of compute-intensive layers such as convolutional, fully-connected, and RNN layers. Multiple institutions from both industry and academia are working together on MLPerf [79], a rich collection of benchmark suites targeting DNNs. Researchers from academia also made significant contributions. Zhu et al. developed TBD [80], a benchmark suite aimed at the training of DNN models for many popular problem domains. Hu et al. developed Mirovia[81], a benchmark suite incorporating many machine learning applications and commonly-used DNN primitives, targeting heterogeneous computing platforms. Karki et al. developed Tango [82], a DNN benchmark suite for resource-constrained accelerators such as mobile GPUs and FPGAs; Gao et al. developed

AIBench [83], providing benchmarks from a wide range of AI applications for high-performance computing platforms.

The development of benchmarking tools and frameworks for DNNs has enabled many researchers to profile and characterize this class of workload. There have been many earlier evaluation studies that focused on neural networks. Shi et al. conducted a series of experiments, evaluating the current state-of-the-art deep learning software tools [84]. They evaluated a number of neural network models using state-of-the-art deep learning tools on both single and multiple GPUs, proposing a general guide to leverage the right software tools on the targeted platforms. They also pointed out possible optimization directions for researchers. Kim et al. also evaluated several existing deep learning frameworks and suggested possible optimization methods leveraging convolution algorithms to improve CNN efficiency [85]. They characterized existing deep learning frameworks at an application level and explored the benefits of using different convolution algorithms to achieve better performance. Rhu et al. measured the memory usage of DNNs and proposed a virtualization method to deal with issues associated with the limited memory on a GPU [86]. They characterized the data access patterns to create a virtual memory management strategy for DNN applications. Mojumder et al. profiled DNN workloads on multi-GPU environments, evaluating the data communication performance among computing nodes, under different data transfer methods [87]. Sun et al. also characterized the DNN workload on AMD GPUs in their work, evaluating performance tradeoffs on the ROCM platform [88]. They conducted a performance analysis of one DNN model on a layer-by-layer basis, leveraging the ALU utilization to shed some light on execution details. Karki performed a detailed analysis of different DNN models on multiple platforms, including a server-grade GPU, a mobile GPU, a simulator and an FPGA [89]. They presented selected performance statistics and power consumption, comparing them across the different platforms.

3.2 Optimization

Prior work has extensively explored optimization methods for scalability, performance, memory usage, and energy efficiency of deep learning applications. Among the existing literature in this area, we have identified three paths pursued by the researchers: i) redesigning a DNN algorithm, reducing the timing complexity of time-consuming DNN primitives, ii) making existing platforms more intelligent and efficient, and iii) developing customized hardware. We also note that researchers sometime pursue a algorithm-hardware co-design strategy, pursuing two paths concurrently. For example, Ding et al. [90] describe the design of a new algorithm coupled with a weight matrix

compressing technique, reducing the timing complexity of solving GEMM. They also proposed custom hardware to accommodate the new algorithm. Yao et al. [91] developed a pruning algorithm to generate sparsity for weight matrices of DNN models, while achieving efficient execution on GPUs for DNN inference. In this section, we only review the work the follows the first two paths. We discuss the related work down path iii), as well as algorithm-hardware co-design, in the next section.

Many previous studies show significant performance improvement by redesigning DNN algorithms. The convolutional layer is the most frequently used primitive type in CNN models, as well as the most timing-consuming layer. Researchers have focused on optimizing the convolutional layer. Fast Fourier Transforms (FFTs) have been widely used to reduce the timing complexity of solving convolution [92]. Winograd [93] is another popular algorithm used to reduce the number of computations required for running a convolution with selected convolution kernel sizes. Besides optimizing the convolution algorithm, two other techniques frequently used to simplify the computation of the convolutional layer and the fully-connected layer (linear transformation layers). The first is to compress the weight matrices, reducing both the storage space and computations; the second is to utilize sparseness of the DNNs, skipping computations involving zeros.

3.2.1 Model Compression

Model compression is a lossy method since compressing weights leads to loss of information. As a result, model accuracy can be affected. However, from the previous research, the impact is usually tolerable, even negligible in some cases. Weight pruning is one model compression technique. Castellano et al. proposed an iterative pruning algorithm that carefully determines the elimination of nodes, keeping the performance from worsen iteratively [94]. Han et al. proposed a three-step mechanism that applies connection pruning, quantization, and Huffman coding, reporting a 9× to 13× improvement in model size, while achieving a 3x-4x speedup and 3x-7x better energy efficiency [62, 63]. Anwar et al. introduced a structured pruning for convolutional layers, demonstrating the effectiveness of structured pruning [95]. Huang et al. presented a learning algorithm to prune filters of DNN with the help of a reward funciton [96]. Zhang et al. developed a systematic weight pruning framework using the alternating direction method of multipliers (ADMM) [97]. Quantization is another popular model compression method. Lin et al. proposed a weight quantization method which identifies the fixed-point representation with the best bit width, achieving a reduced model size without any accuracy loss [98]. Wu et al. proposed a unified quantization framework for CNNs,

reducting model execution time, and reducing the model size [99]. In addition to these two studies, researchers have leveraged low-rank matrices to compress parameter matrices. Jaderberg et al. leveraged low-rank matrices in the convolutional layer, reducing the number of computations [100]. Tai et al. proposed an algorithm to perform low-rank tensor decomposition in convolutional layers, creating low-rank representations for the filter kernels [101]. These approaches offer reasonable parameter reductions with only a minor impact on accuracy.

3.2.2 Exploiting Sparsity

Exploiting tensor sparseness is commonly leveraged to improve the compute efficiency of DNN execution. A tensor is sparse when its sparsity (i.e., the percentage of zero elements) is high. The primary sources for generating zeros during DNN execution are due to weight pruning and ReLU activations. Weight pruning is a class of algorithms that can remove redundant parameters, generating zeros, which reduces the number of computations. Note that weight pruning is also a model compression method, based on its ability to reduce the size of the model, as discussed above. Here, we focus on its benefits in terms of increasing the sparsity of a tensor.

Wen et al. proposed a learning method to regularize the structured sparsity [102]. Liu et al. made use of the sparsity created by pruning the redundancy in weights of a CNN, with the goal of simplifying the computation with only minor model accuracy loss [103]. Narang et al. used group lasso regularization to create sparse blocks of each layer in RNN, improving the overall computing efficiency [104]. Han et al. proposed to manually obtain sparsity and tune training by removing the less essential weights [105]. This work intends to improve model accuracy rather than simplify computation or memory usage. The focus of this work is to improve the prediction accuracy with more model complexity. The ReLU activation function, on the other hand, inherently generates zeros. Therefore, exploiting activation sparsity is lossless. Ren et al. proposed a method to predict structured block sparsity in activations during inference, achieving acceleration on a GPU [106]. Judd et al. discussed different encoding mechanisms for effectual activations and their associated memory footprint [107]. Shi et al. exploited the activation sparsity after ReLU to accelerate the execution of CNNs on CPUs [108].

Accelerating sparse matrix operations, particularly sparse GEMM operations, on GPUs has gained a lot of attention since many different high-performance computing applications can benefit from leveraging sparse GEMM [60]. GPU vendors have developed multiple sparse GEMM libraries, including cuSparse [61], clSparse [109], and rocSparse [110], supporting commonly-used

sparse formats and high-performance sparse GEMM implementations specifically for their GPUs. Previous studies proposed a number of novel sparse formats to achieve efficient execution on GPUs. Liu et al. proposed CSR5, a format that supports efficient sparse matrix-vector multiplication for irregular matrices [111]. Steinberger et al. proposed HOLA-SPMV, a mechanism to achieve efficient load balancing and memory access patterns [112]. Vazquez et al. proposed ELLPACK-R, a format designed based on ELLPACK, with optimized intermediate index vectors and efficient SpMV execution on GPUs [113]. There are also studies investigating accelerating sparse GEMM on GPUs. Liu et al. proposed a general sparse GEMM method for irregular matrices, addressing problems including operation insertion and load imbalance [114]. Dalton et al. proposed an optimization method for sparse GEMM on GPUs. They divided the SpGEMM operation into three phases that can be parallelized, and employed a set of optimizations to improve memory performance [115]. Gray et al. developed a set of GPU kernels that take advantage of block-based sparsity to accelerate DNN on GPUs [116]. Yang et al. proposed a set of principles to design sparse GEMM on GPUs. The work focuses on increasing instruction-level parallelism and achieve better load-balancing [117].

3.2.3 General Optimizations on GPUs

Other than redesigning the DNN algorithms to achieve acceleration, many of the researchers have explored methods that efficiently utilize general-purpose hardware platforms. A straightforward approach is to duplicate a DNN model to multiple computing nodes, enabling distributed DNN execution. Usually, the distributed execution is aimed for DNN training only. For each iteration, the training process requires a mini-batch of data before updating the parameters. The distributed training processes multiple mini-batches simultaneously on different computing nodes. Many deep learning frameworks, such as Tensorflow[76], Caffe[59] and CNTK[118] leverage distributed training for acceleration. Eliuk et al. developed a distributed DNN library to accelerate DNN execution on a GPU cluster [119]. Awan et al. extended Caffe to enable distributed training on a GPU cluster [120]. Mittal et al. [121] summarize some the optimization methods available on a single GPU, including: 1) optimizing the data layout, 2) leveraging data re-use, 3) improving tiling for batched GEMM, and 4) optimizing GPU kernel scheduling. The goal is to exploit the GPU resources efficiently. Li et al. studied the impact of changing the data layout of tensors in DNNs, highlighting the benefits of using selected data layouts to achieve better memory efficiency [122]. Chen et al. optimized the memory access pattern and execution pattern of convolution, taking advantage of the data re-use pattern when performing direct convolution operations [123]. Li et al. proposed a framework to identify

an optimized tiling and batching scheme, identifying an efficient kernel design mechanism [124]. Jain et al. proposed a Just-in-Time (JiT) compiler to coalesce GPU kernels, increasing the resource utilization when performing DNN inference [125].

3.3 Computing Hardware Design

Matrix operations and dot products dominate the computation of DNNs. As such, accelerating DNN execution focuses on designing *custom hardware* for matrix operations and dot products. Representative work include Google's Tensor Processing Unit [126, 127] and IBM's TrueNorth [128, 129, 130]. The TPU, designed by Google, is a custom ASIC hosting 65,536 8-bit MAC matrix multiply units. The peak throughput of a TPU can reach 92 TeraOps/second (TOPS). Custom hardware is also playing a role in the design of general hardware. NVIDIA has added Tensor Cores in the Volta architecture to accelerate DL applications [131]. IBM's TrueNorth incorporates neurosynaptic cores inspired by actual brain activity, integrating both computation and memory, achieving significantly lower power consumption. There have been many other designs proposed. Chen et al. proposed DianNao, a custom accelerator for machine learning algorithms, especially DNNs [54]. Rather than improving compute performance, they focus on designing hardware that can improve memory performance. Chen et al. proposed DaDianNao, a custom computing platform optimizing the dataflow of a DNN. They map the memory footprint to an on-chip storage, significantly reducing off-chip data communications, while achieving high bandwidth [56].

In addition to improving compute and memory performance, many hardware designs leverage model compression and exploit sparsity in both weights and activations. Both techniques create irregularities for DNNs. For example, model compression techniques may introduce other types of computations in addition to matrix operations. The sparsity may change the memory footprint and the locality. Thus, designing custom hardware becomes the primary trend when leveraging model compression and sparsity. Qiu et al. developed a novel CNN accelerator on an FPGA, with a significantly increased degree of parallelism for the convolutional layer. They also leveraged dynamic quantization techniques and Single-Value Decomposition (SVD) to reduce the number of parameters of the fully-connected layers [55]. Cheng et al. studied parameter redundancy in DNNs by using circulant matrices for weight representation of the fully-connected layers [132]. They proposed the corresponding accelerated inference and training algorithms for the fully connected layers. As a follow on work, Ding et al. evaluated the performance versus accuracy tradeoff of using block-circulant weight matrices [90]. They generalized the approach for both fully-connected layers

and convolutional layers and provided a cross-platform hardware design and optimization solution for deep learning systems. Deng et al. proposed the PermDNN architecture on an FPGA, accelerating DNNs by leveraging permuted diagonal matrices [133]. Turner et al. [134] and Yu et al. [135] show that generating sparsity in the network weights can hurt inference performance, which is one of our motivations for designing specialized accelerators. For example, Han et al. [43] took advantage of pruned and quantized weights to design an energy-efficient inference engine for embedded systems, which was evaluated on nine DNN benchmarks. This work targets fully-connected layers, though only accounts for roughly 10% of the overall computation in modern DNNs. *Cnvlutin* [136] is an accelerator that follows a value-based approach to dynamically eliminate ineffectual multiplications (those that include an operand that is zero). *Minerva* [57] exploits the observed network sparsity to minimize data accesses and MAC operations, enabling low-power acceleration for highly-accurate DNN prediction. *Eyeriss v2* [137] proposes an accelerator architecture designed for running compact and sparse (in the weights and activations) DNNs. Han et al. [44] and Ren et al. [138] present efficient speech recognition engines that can work directly on a compressed sparse LSTM model on an FPGA. Jain et al. [64] proposed *Gist*, a strategy for exploiting the sparsity of selected intermediate activations to achieve efficient data compression for saving memory space. Their focus is on optimizing memory usage. They incur a 4% performance overhead. Dey et al. [139] and Cao et al. [140] present reconfigurable hardware architectures for accelerating training, which use pre-determined and structured sparsity to lower memory and computational requirements. Finally, Rhu et al. [141] present a general-purpose compression DMA engine for high-performance DNN virtualization. The virtualization strategy uses the CPU's memory to store the training data and improve the overall training performance by allowing more training data on a GPU. Qin et al. [142] proposed *SIGMA*, a flexible and scalable accelerator architecture, adopting a novel reduction tree microarchitecture to accelerate irregular sparse matrix operations. However, SIGMA targets custom accelerator architectures rather than GPUs.

3.3.1 Near-Memory Processing

Memory efficiency is one major bottleneck encountered when running a DNN [54, 56]. One approach to optimize memory performance is to add a special-purpose processing unit near memory to achieve high-performance and low-latency data management. In the early 2000s, IBM proposed MXT technology to perform real-time compression and decompression of data to reduce traffic between the shared cache and the main memory [143]. Recently, NVIDIA added an L2 compressor

as a new feature in CUDA 11, supporting compression on pinned GPU memory to reduce traffic between L2 and DRAM [144]. Many researchers have employed near-memory processing units to accelerate systems with DNNs. Kwon et al. proposed TensorDIMM, a mechanism involving a custom DIMM module equipped with near-memory processing cores tailored for DNN operations [145]. Ke et al. proposed a lightweight near-memory processing solution to accelerate a recommendation system. They employed techniques such as memory-side caching, table-aware packet scheduling, and hot-entry profiling, achieving significant performance gains [146]. Schuiki proposed NTX, a near-memory acceleration engine for DNN training at scale. The engine significantly reduces offloading overhead and supports an optimized data path for convolution and gradient aggregation. Their proposed system comes equipped with RISC-V cores and an NTX engine, outperforming a GPU-based system in terms of execution time and energy efficiency [147].

3.4 Summary

In this chapter, we reviewed related research on benchmarking. We presented the popular benchmark suites that are targeted for GPUs, highlighting their compute pattern coverage and purposes. We also presented a number of optimization studies for DNNs. The first class focused on redesigning DNN algorithms, including improving the convolution algorithm, reducing the model size, and exploiting sparsity. The other class explored methods utilizing existing platforms intelligently and efficiently. Finally, we presented previous work that pursued new hardware designs. Given the inefficiency of running DNNs on existing hardware platforms, researchers have proposed many custom hardware designs, improving both compute and memory performance. Some proposed architectures exploited model compression and sparsity.

Chapter 4

DNNMark: A DNN Benchmark Suite for GPUs

Deep learning algorithms and applications are quickly becoming the primary focus of attention for many leading research and industrial labs. Currently, a large number of researchers from both academia and industry are expending significant effort to develop new DNN models. In order to develop a more robust DNN model that has higher capacity, researchers keep increasing the number of in-network parameters, increasing the depth of the network by adding more layers, which results in training that can take days or even weeks to complete.

Given that the computations of each layer in the deep neural networks are matrix-based, DNN computations can significantly benefit from using computing platforms that can exploit data-level parallelism. Training a DNN model requires an enormous amount of training data, which has been a fundamental barrier to leveraging DNNs. Traditional computer systems (e.g, CPUs) are not able to provide enough computational power to tackle the training task. Thus, heterogeneous computer systems composed of both CPUs and GPUs are becoming the defacto standard platform for training DL algorithms. When using a system equipped with GPUs, the computation associated with each layer in the DNN is offloaded to the GPUs.

However, if we want to tune the performance of DNN computations on a GPU platform, there is a void of tools available. While there are many existing deep learning frameworks (e.g., Caffe[59]) that provides GPU support in order to shorten the training time, they focus on reducing the programming effort versus tuning performance. This prior work paid little attention to tuning the software performance. Existing frameworks cannot satisfy the needs of designers of next-generation

hardware platforms, who desperately need a framework where they can evaluate design tradeoffs when executing DL algorithms. While some researchers have considered extracting DNN primitives and embedding them in reconfigurable hardware, this approach is too labor-intensive. Furthermore, configurability will be compromised by a customized solution. We still need to perform model training, the overhead of which can vary significantly based on the platforms available.

In this chapter, we present DNNMark, a benchmark suite designed to address many of the problems discussed above. First, each of DNN primitive workloads can be easily invoked separately, without any sacrifice on configurability. One can specify any type of scenario for benchmarking on an algorithm-specific level, and configuring such parameters is no longer an issue in this suite. Second, the actual measurement can be done during the execution of any specific kernel. Our framework allows us to ignore the setup stages and focus on only the computation steps.

Unlike other DL frameworks, attaching a real database for training purposes is not mandatory anymore. This capability will significantly aid the computer architecture community, which is more interested in designing/tuning hardware/software, and less interested in the details or configuration of DNNs.

Depending on the specific configuration, DNNs can involve combinations of DNN primitives. A model composed of two or more primitive functions may be more desirable during performance evaluation. In such cases, a composed model rather than standalone primitives, are preferred. To provide this capability, DNNMark can be extended to model more sophisticated DNN models, where layers are connected to, and dependent upon, each other.

4.1 Software Design

DNNMark is developed in C++ and uses external libraries including cuDNN and cuBLAS for NVIDIA GPUs, and MIOpen and rocBLAS for AMD GPUs. We utilize the CMake tools to enable effortless compilation and build of the framework, and link Google libraries such as gflags [148] and glog [149] for ease of configuration and improved debugging. Unlike other GPU benchmark suites, e.g., Rodinia[69], where each benchmark is developed and built individually, DNNMark includes benchmarks that can be combined into a compound workload, with dependencies indicated between different benchmarks, which allows us to construct an actual DNN model. So that, besides running benchmarks separately, more complex scenarios that combine multiple primitives can be evaluated.

To support this benchmark architecture, we use a different strategy to generate benchmarks in this suite. Instead of developing each benchmark individually, we first develop a centralized

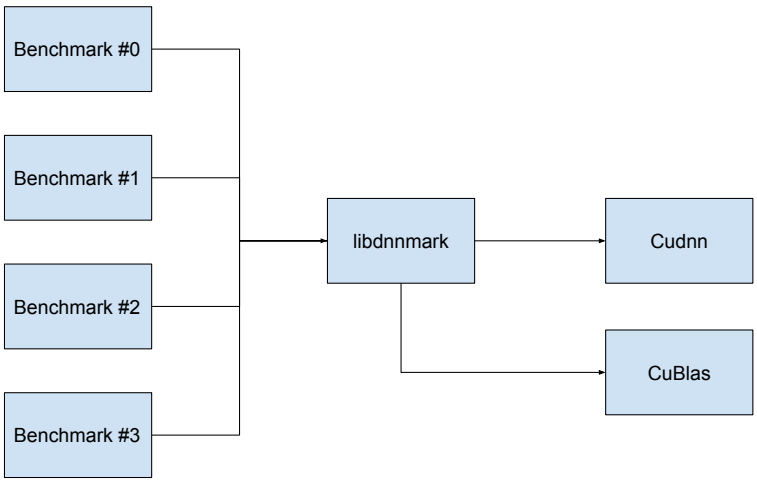


Figure 4.1: Linkage graph of DNNMark.

DNNMark library, which encompasses all required functionalities of the DNN primitives. The library also provides a general easy-to-use interface for various tasks (e.g., configuring the system, initializing the system and running computations). Figure 4.1 shows the overall benchmark flow using our DNNMark library.

As indicated in Figure 4.1, we first build up the DNNMark library *libdnnmark* with external DNN libraries. When this is complete, several benchmarks can be built through linking to this library. Following these steps, benchmark development can be simplified, and DNN-specific development can be managed much more straightforward.

The centralized DNNMark library has a well-established architecture, where the software workflow is well defined. As shown in Figure 4.2, the framework has several software modules that handle different tasks and can also work cooperatively as a whole. The *Config Parser* is in charge of interpreting information from the external configuration files and adding layers based on the configuration parameters to the layer pool, and a data structure that holds all of the configured layers supporting various DNN primitive types. The primary role of the *Initializer* is to set up the tensors and the descriptors based on specified parameters through several encapsulated utility classes in DNNMark. And the Runner module is the one that launches computations of the DNN primitives. As presented in the figure, there is also a Data Manager module that interacts with the layer pool. It is a functional software class that maintains and manages data chunks in GPU memory. Details of this mechanism will be explained later.

From Figure 4.2, we can see the specific execution order, marked with numbers, in

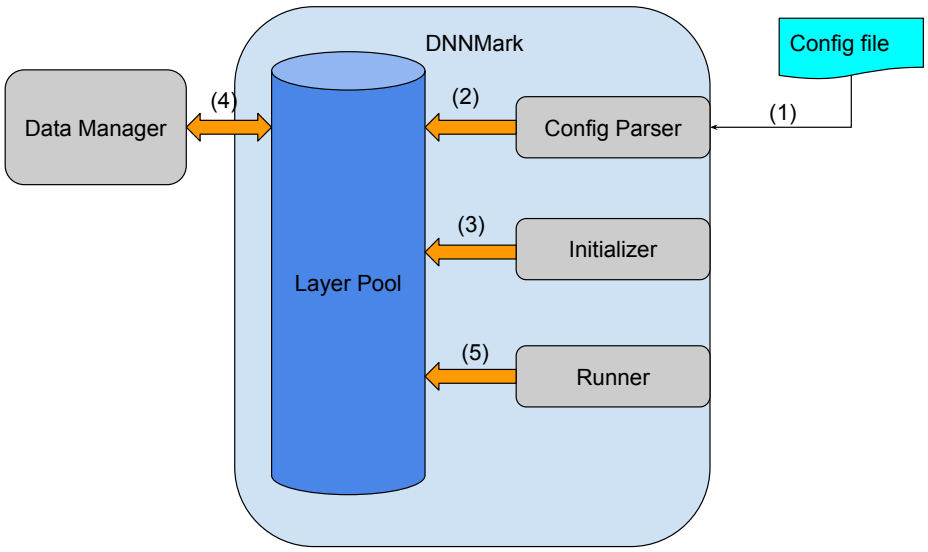


Figure 4.2: The software architecture of DNNMark.

the workflow. More specifically, once the DNNMark framework is started, we first access the configuration file and locate the target sections in order to extract the parameters. Next, one or more layers are added to the layer pool, and initialization starts. During the initialization phase, configured layers request data chunks from the Data Manager and keep track of them. Finally, with model construction complete, the actual computations can be started for benchmarking.

As mentioned previously, one of the most critical functions in DNNMark is data management. The main job of the Data Manager is to maintain and manage data chunks in GPU memory, as well as interact with layers in the layer pool. In order to obtain memory space for input/output or parameters, the layers can request GPU data chunks of a specific size and then the Data Manager will return the Data IDs corresponding to the requested GPU data chunks. With these IDs, layers are able to assign the addresses of the allocated memory spaces. This design not only liberates layers from performing data allocation but also provides a clean data management mechanism to create composed models, where some data chunks are shared between layers. Figure 4.2 provides an example of the Data Manager module. As shown in Figure 4.3, the three layers are assigned with unique Data IDs, which can be further used to locate the actual memory space in GPU memory. Given this design and implementation, DNNMark is highly configurable, easy to use, and highly extensible.

Configurability: Following the workflow presented earlier, the first step of initiating a benchmark is to read a self-defined DNNMark-specific configuration file. The file contains all the

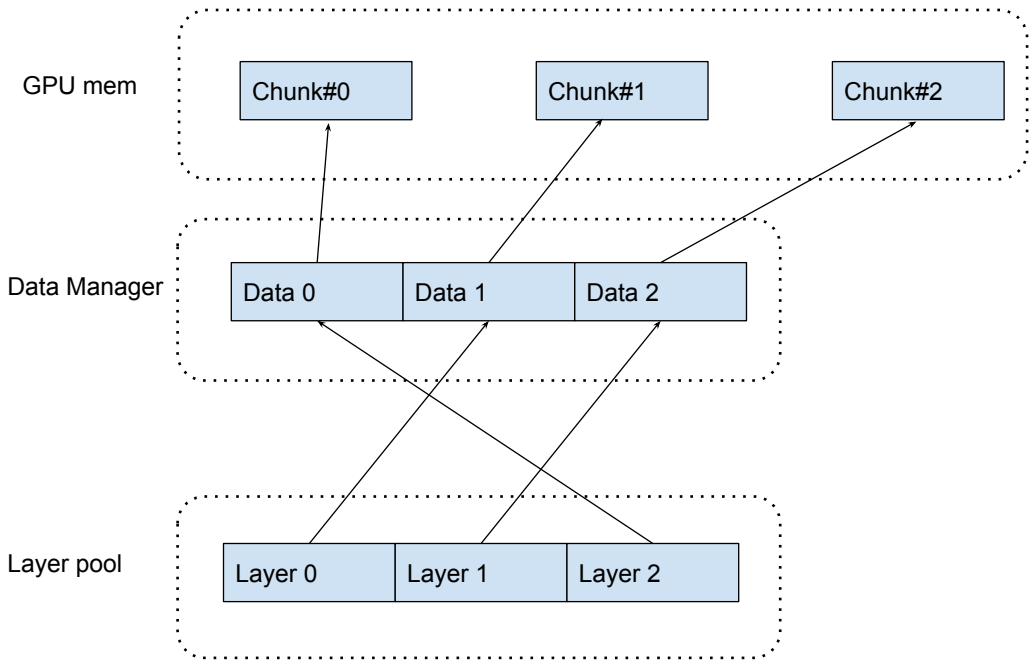


Figure 4.3: Data management mechanism of DNNMark.

configurable parameters. In this step, both DNNMark and the layer-specific parameters will be extracted. List 4.1 shows an example of a configuration file.

Listing 4.1: Example of a configuration file.

```
[DNNMark]
run_mode=standalone

[Convolution]
name=conv1
n=100
c=3
h=256
w=256
previous_layer=null
conv_mode=cross_correlation
num_output=32
kernel_size=5
```

```
pad=2
stride=1
conv_fwd_pref=fastest
conv_bwd_filter_pref=fastest
conv_bwd_data_pref=fastest
```

As shown in the configuration file, there is one general parameter which defines the framework running mode. The next field specifies one convolution layer, and includes both general and algorithm-specific parameters. The general parameters include those regarding layer names, data, and kernel dimensions, while the algorithm-specific ones specify convolution mode and algorithmic preferences for running forward and backward propagation. As we can see, allowing for configurable algorithm-specific parameters in this framework enables us to explore a wide variety of GPU execution patterns in a more flexible way. Besides, another aspect of this flexibility is that when explicitly indicated, a benchmark can run a subset of the DNN primitives without the need to specify a new configuration file.

Convenience: We provide a straightforward method to construct DNN-related benchmarks thanks to the centralized design scheme of DNNMark. Our framework significantly shortens benchmark development time and simplifies the procedure of adding additional DNN primitives. For example, the data management and utility functions can be reused so that programmers only need to concentrate on functional elements when adding a new primitive. Currently, we support seven different DNN primitives that are essential parts of composing commonly-used DNN models such as AlexNet [16]. But in the future, more primitives can be added easily.

Extensibility: As shown in list 4.1, DNNMark allows the user to specify the run mode as a framework-specific configuration parameter. Presently, we support two modes: standalone and composed. In standalone mode, benchmarks run in an isolated manner, meaning that only one DNN primitive is invoked at a time. However, under certain circumstances, running a composed model, where multiple DNN primitives are connected to represent a full network model, could be more interesting. In a composed mode, the user can add a single DNN primitive to a new benchmark. The procedure is as simple as editing a configuration file. List 4.2 shows another example of a configuration file with the running mode set to ”composed”.

Listing 4.2: Example of a configuration file using composed mode.

```
[DNNMark]
```

```
run_mode=composed

[Convolution]
name=conv1
n=1
c=3
h=32
w=32
previous_layer=null
conv_mode=cross_correlation
num_output=32
kernel_size=5
pad=2
stride=1
conv_fwd_pref=fastest
conv_bwd_filter_pref=fastest
conv_bwd_data_pref=fastest

[Pooling]
name=pool1
previous_layer=conv1
pool_mode=max
kernel_size=3
pad=0
stride=2

[FullyConnected]
name=relu1
previous_layer=pool1
num_output=32
```

In this example, there are three layers, Convolution, Pooling, and Fully Connected, linked together to form a new model. Other than changing the run mode, the ”previous_layer” parameter needs to be

defined correctly in order to run properly.

4.2 Supported DNN Primitive

Next, we present the supported DNN primitives and models in DNNMark. Table 4.1 lists the details.

Table 4.1: Supported DNN primitives and models in DNNMark.

DNN primitives and models	Description
Convolution	2D Image convolution (chapter 2)
Pooling	Down-sampling techniques (chapter 2)
Local Response Normalization (LRN)	Normalization technique inspired by lateral inhibition (chapter 2)
Batch Normalization (BN)	Normalization technique to achieve fast training converge (chapter 2)
Activation	Non-linear activation functions (chapter 2)
Fully-Connected Layer	Traditional neural network model (chapter 2)
Softmax	Function of output layer (chapter 2)
AlexNet	[16]
VGG16	[17]

Both cuDNN and MIOpen provide the implementation of the above DNN primitives, with similar API interfaces and options. We use the phrase ***DNN library*** to represent cuDNN and MIOpen. In DNNMark, we provide multiple algorithm options for convolution to capture the different behaviors of the different algorithms flexibly. The FFT and Winograd are fast algorithms that are widely used [49]. The former requires significant memory space, while the latter is memory-efficient[150]. Our selected DNN library also allows the user to select the execution parameters in terms of speed and memory usage, correspondingly recommending either a performance-oriented or a memory-efficient algorithm. Likewise, DNNMark supports flexible parameter configurability for all other DNN primitives. In the configuration file, the user can specify the parameters for each DNN primitive type. For example, the Pooling has a different configurable mode, e.g., MaxPooling and AvgPooling. LRN has configurable parameters such as k, α, β, N, and n, as indicated in equation 2.7.

Table 4.2: Nvidia K40c Configurations

Type	k40
Number of processor cores	2880
Warp size	32 threads
SIMD lane width	8
Maximum threads per block	1024
Number of 32-bit registers	65536
Maximum registers per threads	255
Configurable shared memory/L1 cache	64KB
Read-only data cache	48KB
L2 cache	1536KB
Memory interface	384-bit
Memory bandwidth	208 GB/sec
Memory size	12GB

Table 4.3: Library Details

Libraries	Version
CUDA	8.0
CuBlas	8.0
CuDNN	5.0

The activation function has multiple configurable options, e.g., ReLU, Sigmoid, and Hyperbolic Tangent. The softmax has different configurable algorithms and modes. We use the BLAS library for the fully-connected layer, cuBLAS for NVIDIA GPUs, and rocBLAS for AMD GPUs.

4.3 Evaluations

4.3.1 Environment Setup

We select the Nvidia K40c [151] micro-architecture from the Nvidia Kepler family of GPUs [152] to demonstrate the utility of our benchmarks. Table 4.2 describes the hardware details.

The library details can be found in table 4.3.

Table 4.4: Selected benchmarks

Benchmark	Batch size	Number of Channel	Height	Width
Convolution	100	3	256	256
Max Pooling	100	3	256	256
LRN	100	32	256	256
Activation(ReLU)	100	32	256	256
Fully connected	100	9162	1	1
Softmax	100	1000	1	1

Table 4.5: Selected kernels of benchmarks

Benchmark	Forward Kernel	Backward Kernel
Convolution	implicit_convolve_sgemm	wgrad_alg0_engine
MaxPooling	pooling_fw_4d_kernel	pooling_bw_kernel_max
LRN	lrn_fw_Nd_kernel	lrn_bw_Nd_kernel
Activation(ReLU)	activation_fw_4d_kernel	activation_bw_4d_kernel
Fully connected(FC)	sgemm_sm35_ldg_ nt_128x16x64x16x16	sgemm_sm35_ldg_ nt_128x16x64x16x16
Softmax	softmax_fw_kernel	softmax_fw_kernel

4.3.2 Selected Benchmarks

In our evaluation, we chose to evaluate individual DNN primitives which include benchmarks that are based on a subset of the layers in AlexNet. Table 4.4 lists all the selected benchmarks and the dimensions of their inputs. Note that for each benchmark, both forward and backward propagation are evaluated.

For some of benchmarks, there is more than one kernel in the corresponding API function. In order to simplify the evaluation and comparison between benchmarks, we selected a collection of representative kernels, as shown in table 4.5.

4.3.3 Results

We use the nvprof[153] profiling tool and present results based on several selected metrics provided by the tool.

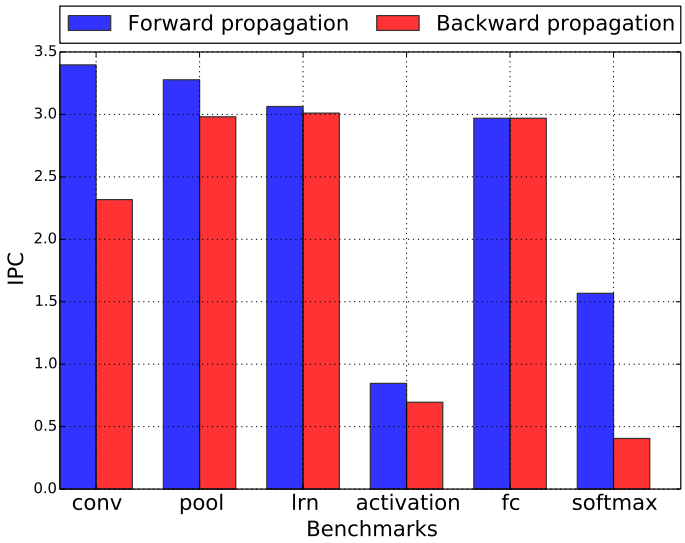


Figure 4.4: Instructions per cycle achieved.

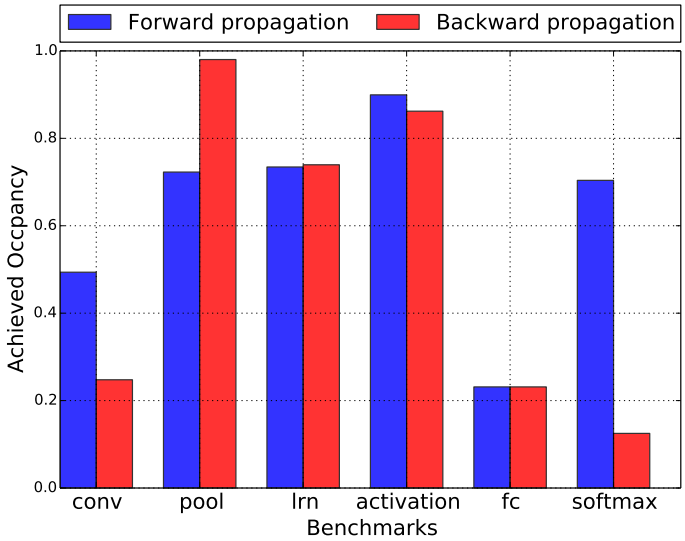


Figure 4.5: The occupancy achieved.

Figures 4.4-4.6 present the execution Instruction Per Cycle (IPC), occupancy, and eligible warps per cycle for six different DNN primitives, with both forward and backward propagation. IPC provides us with a metric to measure instruction throughput and performance. Achieved occupancy reflects, to some extent, the current thread activity. The number of eligible warps per cycle reflects the effectiveness of the warp scheduler. Additionally, Figure 4.7 shows the GPU utilization of different

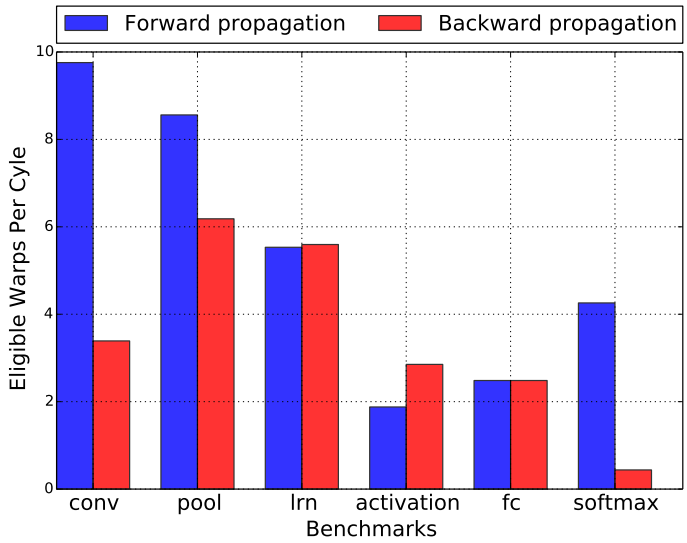


Figure 4.6: The number of eligible warps per cycle.

hardware resources, including: L1/shared memory, L2 cache, DRAM, the ALU and the Load/Store Unit.

From these profiling results, we can see the detailed program behavior of each benchmark. For example, from the IPC results, we find that activation has relatively poor IPC for both forward and backward executions. From Figure 4.7, we see that activation has low ALU utilization, but high utilization of DRAM and the Load/Store Unit. The occupancy also captures this behavior, because lower occupancy implies lower thread activity. From these observations, we can conclude that the execution of activation is memory-bound. From the IPC results in Figure 4.4, we see that convolution, pooling, LRN and the fully-connected layer all have high IPC. They have high ALU utilization, so we can conclude they are compute dominated. But one interesting observation is that even though backward pooling has significantly high occupancy, the IPC is in fact not as high as forward pooling.

4.4 Discussion

A DNN is not just a standalone application, but may also be an element of a larger application. Given this particular structure, optimizing a single computation is no longer a solution for deep learning algorithms. The entire problem has to be considered as a whole. Nevertheless, treating deep neural networks as a complete black-box system also hinders the possibility of exploring potential techniques to leverage current hardware. The recent solution to accelerate training of a

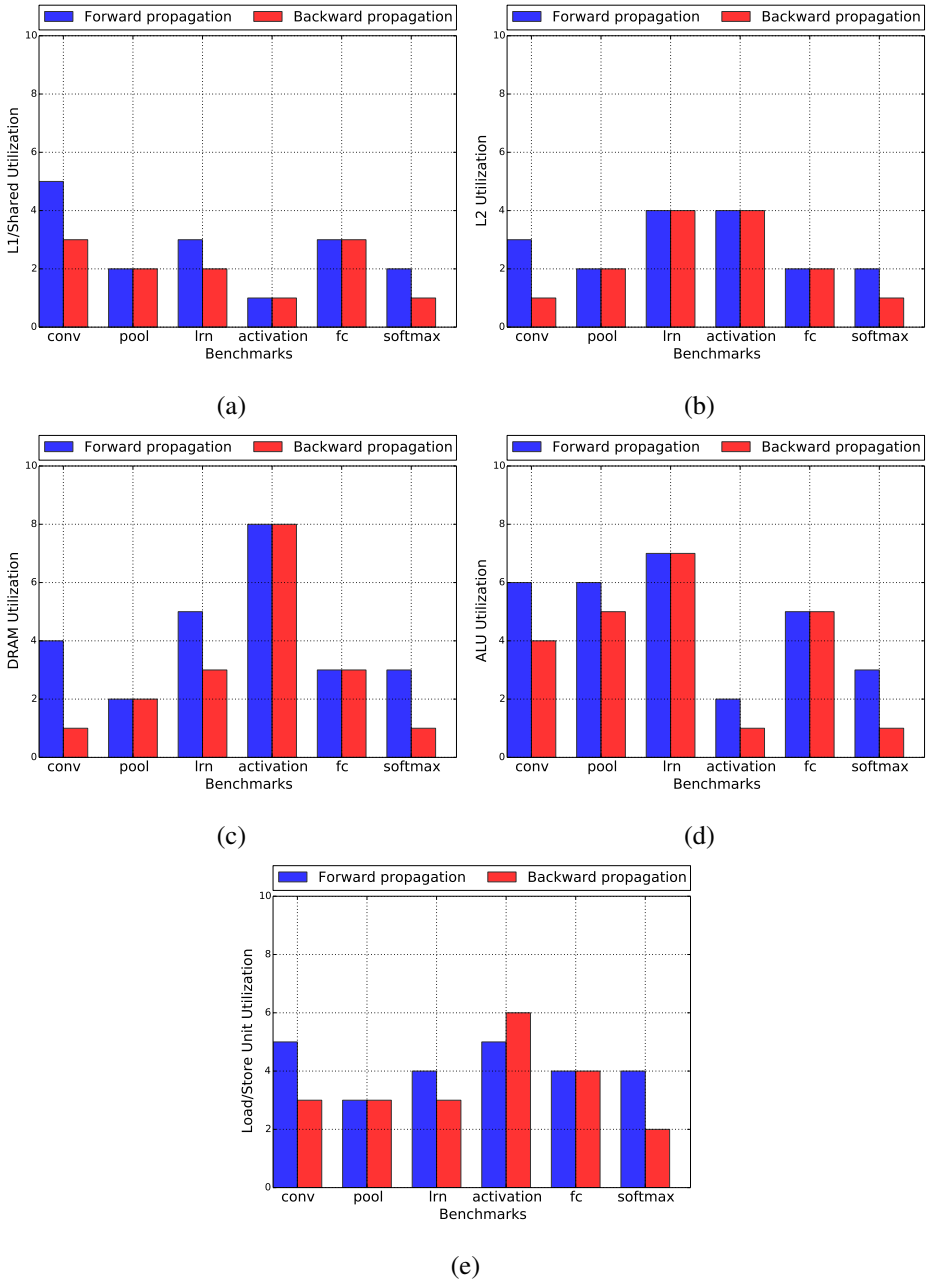


Figure 4.7: A breakdown of GPU utilization.

DNN is still limited to adding more expensive hardware, building large-scale distributed systems and computing clusters[154].

HPC solutions to deep learning are effective. Weeks of training can be reduced to days or even just hours. But the price is also high. The question is whether learning applications are

making efficient use of current hardware. In order to answer this question, a complete profile of each primitive in a DNN needs to be performed. Only with the detailed hardware execution information can computer architects begin to reason about new architectural features to support this workload. Instead of throwing more hardware at deep learning applications, we feel it is a good time to take a step back and explore the potential for new architectural features.

Since a GPU has impressive computional power and high-bandwidth memory to run the training phase of DL, it has become an essential hardware platform for many deep learning systems. Given that the GPU has a significantly larger number of compute resources than traditional computer architectures, we need to make efficient use those massive resources. Equipped with DNNMark, architects can now carry out a wide range of studies to improve performance and power. We plan to explore newer features on more recent GPUs, including concurrent kernel execution [152] and dynamic parallelism [152].

Chapter 5

Characterization of a CNN Execution on GPUs

A Convolutional Neural Network (CNN) is one popular variant of a DNN that leverages convolution as its major linear transformation for feature extraction. It has been demonstrated that a CNN can be very effective in vision and speech classification domains [16, 50]. Similar to other models of deep neural networks, the essential computations in CNNs have been accelerated using a GPU, especially given that most computational operations involving convolution are matrix-based. Since GPUs are an effective target for accelerating matrix operations, they have been quickly developed into a key resource for accelerating CNNs. However, there has only been limited prior work with regards to the execution of this type of workload on GPUs. Given this limited knowledge, it becomes challenging to optimize GPU architectures to run this class of compute-intensive applications.

In this chapter, we capture and analyze the micro-architectural information from two GPUs. The two GPUs are of different product families, a server GPU (Tesla K40) and a desktop GPU (GTX1080), and from different microarchitectural families, Kepler and Pascal, respectively. We study the microarchitectural implications of efficiently running CNNs. To begin this study, we have selected to study AlexNet [16], which is a popular CNN model. Even though the AlexNet is not the latest state-of-the-art CNN model, it covers most of the commonly used primitives in deep neural networks. More importantly, AlexNet's structure is simple to evaluate and its execution presents a number of challenges to current GPU designs. In terms of an implementation of AlexNet, we utilize DNNMark [68] to drive this study, providing a highly configurable and light-weight infrastructure

that builds its core functionality using cuDNN [155] and cuBLAS [156]. These two highly optimized libraries have been used in many DNN frameworks that leverage Nvidia GPUs. Considering their performance, cuDNN and cuBLAS provide a rich software core for us to study DNN execution, working at a microarchitectural level. In our evaluation, we report on execution performance, memory behavior and resource utilization. Furthermore, we identify some of the major limiting factors in GPU microarchitectures when executing DNN primitives.

Based on our results, we first characterize the performance trends of the workload on the two GPU platforms and identify the characteristics of each layer. Then we show that the convolution layer are the main bottleneck during execution of a CNN. From a microarchitectural perspective, we also identify additional limiting factors in the convolution layer, including hardware limits, bandwidth of texture cache. Other than that, we can improve the performance of CNN model with little or even no source code modification. Given challenges in the cache hit rate across all of the layers, we can optimize some layers by bypassing the L1 cache. When L1 cache bypass is enabled, the backward propagation of one convolution layer can achieve a 6.2% speedup on GTX1080. Additionally, we propose a kernel fusion method in which the kernels from the linear data transformation and non-linearity layers can be combined to reduce unnecessary memory transactions without introducing too much extra computation. We have constructed an experiment and observed that the entire DNN model execution is accelerated by 4% on GTX1080. We also discuss the results accordingly.

Although the convolution layer is the main limiting layer in convolutional neural networks, we notice a trend that as hardware advances with many optimizations specifically proposed for linear transformations such as convolution, the other layers start to contribute more in execution time. Hence, a thorough characterization of each primitive should be carried out in order to understand the overall execution behavior of CNNs for future optimization.

5.1 Evaluation Methodology

5.1.1 Workload

In this chapter, we select the AlexNet model to drive our characterization study. Although it is not the latest CNN model, it provides an organization that lends itself to evaluation, while including almost all of the primitives widely used in current state-of-the-art CNN models. Therefore, our microarchitectural characterization when running Alexnet can serve as a representative CNN model. Figure 5.1 shows the organization of AlexNet. As shown in this figure, AlexNet consists of 5

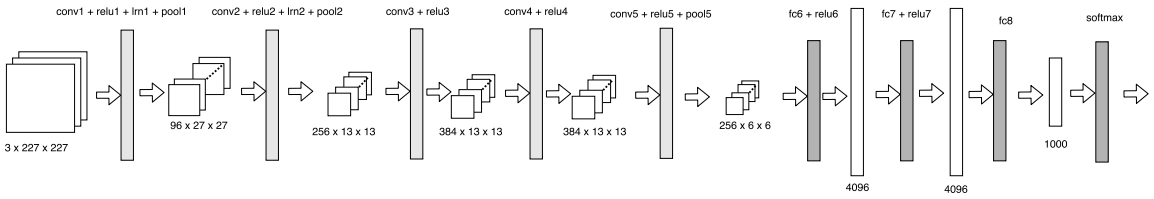


Figure 5.1: The organization of AlexNet.

convolution layers, 3 fully-connected layers, 3 maxpooling layers, 2 LRN layers, 7 ReLU activation layers, and 1 softmax layer. The number of operations in each layer is listed in Table 5.1. Note that we count every occurrence of either arithmetic or logical operations.

In terms of workload, we select cuDNNv6[155], a highly optimized DNN library specifically designed to run on Nvidia GPUs. This implementation has been used extensively by the deep learning research community, since it provides a user-friendly interface and is able to achieve high performance in terms of execution time. We use DNNMark [68], a configurable DNN benchmark suite composed of both cuDNN and cuBLAS, to construct the AlexNet model. Unlike applications found in other popular DNN frameworks, the AlexNet benchmark constructed within DNNMark is designed specifically for measuring hardware performance, essentially reducing the benchmarking effort by removing the need to develop new code. For input, we use a set of synthetic images, generated in batches, with the same dimensions as shown in the figure 5.1.

5.1.2 Hardware

In this chapter, we select the Nvidia Tesla K40 [151] and the GTX1080 [157] as the hardware platforms to run our experiments. The K40 microarchitecture was developed as part of Nvidia Kepler family of GPUs [152], while the GTX1080 is part of the Pascal family. They represents different product grades, as well. K40 is designed for servers, while the GTX1080 is designed for desktop acceleration. These two platforms have different architectures and computing capabilities, serving as good candidates to capture performance trends, while migrating the same workload from one platform to another. Table 5.2 provides details about each device.

5.1.3 Profiling Tools

Capturing and parsing the micro-architectural information of CNNs has many challenges due to the limitations of the Nvidia profiler, nvprof[153]. These issues include: i) some layers

Table 5.1: Number of operations in each layer of AlexNet.

Layer	Number of Operations	Layer	Number of Operations
conv1	210M	conv4	448M
relu1	290K	relu4	65K
lrn1	4M	conv5	299M
pool1	630K	relu5	43K
conv2	896M	pool5	83K
relu2	186K	fc6	75M
lrn2	3M	relu6	4K
pool2	389K	fc7	33M
conv3	299M	relu7	4K
relu3	65K	fc8	8M
softmax	1M		

invoke the same GPU kernel, but specify very different kernel template arguments, meaning that even though the kernel names are identical, they are in fact different kernels. If we want to capture the layer-specific information, using the kernel name only is not sufficient to uniquely identify the kernel; ii) some layers launch the exact same kernel based on their invocation order, according to the network model, so profiling using only the kernel name will return average results for these layers. Therefore, we need to take the invocation order into account so we can capture the information of each individual layer.

This imposes challenges to tie the characterized microarchitectural information to a specific layer. In order to address this challenge, we designed a database-backed trace tracking system that can capture the microarchitectural information for each layer in the CNN model. We establish a relational database to store platform-specific information, providing indices including the layer ID, kernel name, invocation order, and etc. With the help of this trace tracking system, we are able to obtain layer-specific micro-architectural information in a convenient and accurate manner. Figure 5.2 shows the overall workflow of the trace tracking system. We first profile the general execution information, i.e. kernel name and invocation order to create the relational database table, and then we leverage the database to profile the layer-specific microarchitectural metrics and extract the necessary information to drive our analytical tools.

Table 5.2: Nvidia Tesla K40 and GTX1080 configuration details.

Type	Tesla k40	Pascal GTX 1080
Number of processor cores	2880	2560
SIMD lane width	8	
Maximum threads per processor	2048	
Maximum threads per block	1024	
Number of 32-bit registers	65536	
Maximum registers per threads	255	
Shared memory	64KB shared	96KB dedicated
L1 cache	64KB shared	64KB shared
Read-only data cache	48KB dedicated	64KB shared
L2 cache	1536KB	2048KB
GPU maximum clock rate	745Mhz	1607MHz
Memory clock rate	3004Mhz	10000MHz
Memory interface	384-bit	256-bit
Memory bandwidth	208 GB/sec	320 GB/sec
Memory size	12GB	8GB

5.1.4 Experimental Setup

Our experimental framework is designed to capture microarchitectural information at a kernel level for each layer involved in computing a single iteration during the AlexNet training process, without applying SGD. Thus, we focus on the execution of forward and backward propagation. A full evaluation of SGD during training is future work. Although hundreds of thousands iterations will be involved in a complete training, we believe the evaluation of one single iteration can be generalized given that performance metrics are measured at a kernel level, and for each iteration, the same kernels are executed. We use our database-backed trace tracking framework to capture the information from the GPU kernels launched. We run the same experiments with various batch sizes. The number of images in one batch is 16, 64, and 128, which are typical batch-size configurations used in practice.

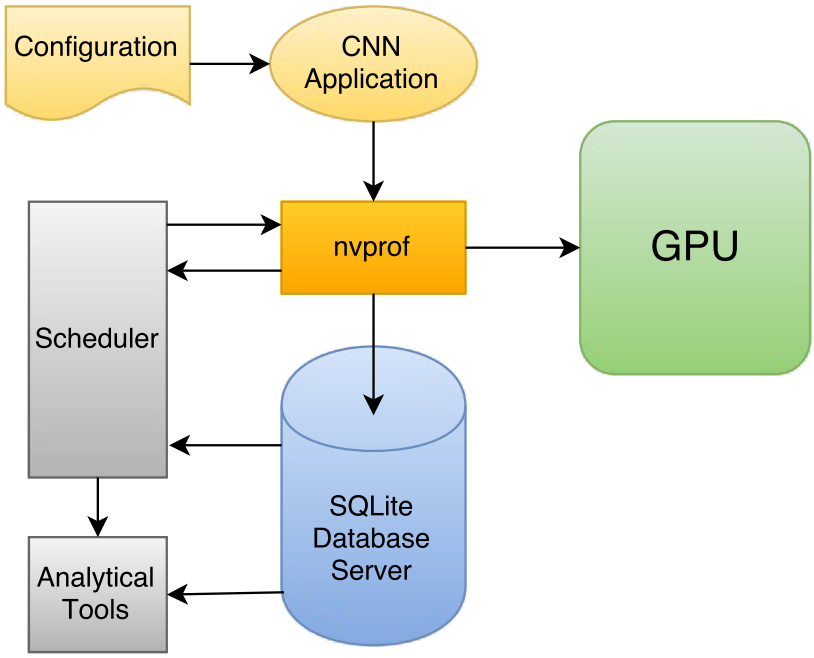


Figure 5.2: Database-based trace tracking system.

5.2 Evaluation results

In this section, we present several key metrics that capture performance in terms of microarchitectural details. Our evaluation is done on a kernel basis. Considering that cuDNN uses a flexible strategy to instantiate kernels while varying template arguments for optimization purposes, we use the layer name rather than kernel name to present our evaluation results, even though the results are measured for kernels of primitives in cuDNN and cuBLAS.

5.2.1 Performance Analysis

First, we evaluate the runtime of each layer involved in one epoch of AlexNet, running across various batch sizes. This gives us a overview of performance for each layer, allowing us to identify the important steps during model execution. Since both the size of input image, as well as the training parameters, are fixed in the model, the batch size becomes the only variable that controls the scale of the final workload and size of the intermediate data. Figures 5.3 and 5.4 showcase the run times of AlexNet running on the K40 and GTX1080, respectively. During backward propagation, layers with trainable parameters should have at least two computations performing both data and backward propagation of weights. Bias is not considered in this chapter, since the related

computations are very simple.

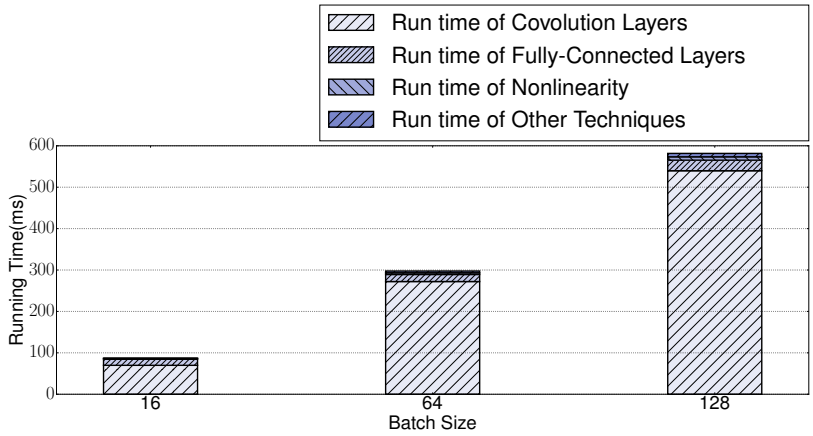


Figure 5.3: Runtime of AlexNet on the K40.

From Figures 5.3 and 5.4, we can clearly see that the layers performing linear transformations are the major bottlenecks during the execution of the entire AlexNet model on both platforms. Convolution layers dominate performance of the linear transformations. This trend is consistent across both platforms, which shows that using a larger batch size leads to better throughput in terms of image processing, meaning that the both platforms achieve good scalability. The execution time is drastically reduced on the GTX1080, though the runtime of the other layers (other than the convolution layers) tends to become more prominent. Moreover, we noticed that the execution time of each layer is well-correlated with the number of operations indicated in Table 5.1. All of the linear data transformation layers take longer to finish.

In Figure 5.5 we report the speedup of running AlexNet on a GTX1080, using the K40 performance as a baseline. Generally, the GTX1080 has more SMs, (although there are fewer CUDA cores in each SM) and a higher clock rate in terms of both processing cores and memory, so the speedup is expected. But it can be observed that the performance gain for each layer varies. The convolution, fully-connected and pooling layers have significantly higher speedup than the activation (ReLU) and softmax layers. The LRN layer has relatively higher speedup during backward propagation versus forward propagation. Based on this observation, we find that the more advanced hardware has a varied impact on the different layers while the floating-point instruction counts are basically equal in the applications built for each platform. The floating-point instruction counts of layers in backward propagation are listed in Figure 5.6. Note that we only present the results from backward propagation because we noticed that the metric trends for the forward and backward

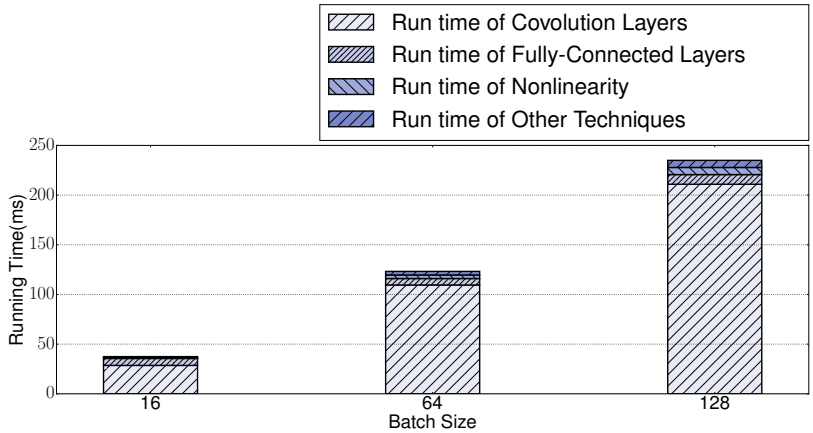


Figure 5.4: Runtime of AlexNet on the GTX1080.

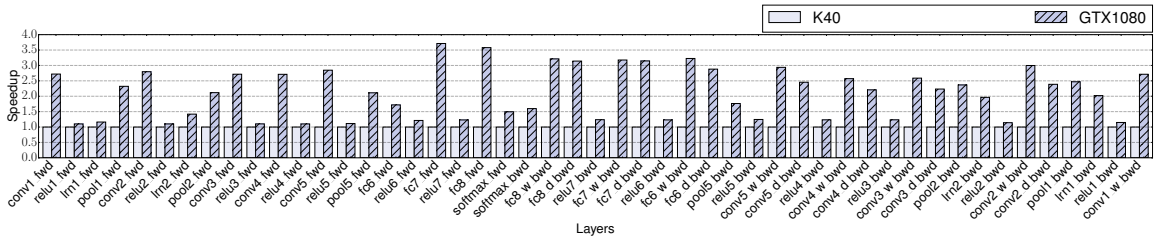


Figure 5.5: Speedup of running AlexNet on the GTX1080, using K40 performance as the baseline.

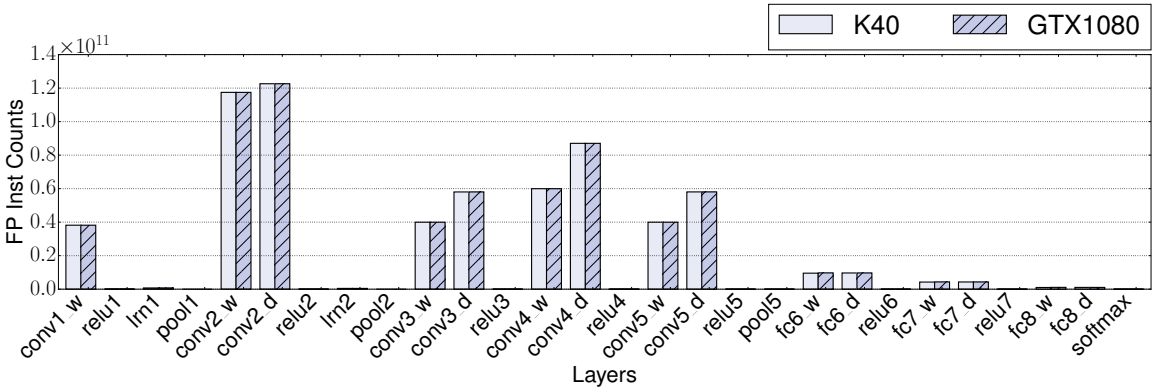


Figure 5.6: The floating-point instruction counts.

propagation across every layer are very similar in most cases, and backward propagation is the most critical part during the CNN training. We present results for a batch size of 128, since that this configuration can fully utilize the massive hardware of a GPU.

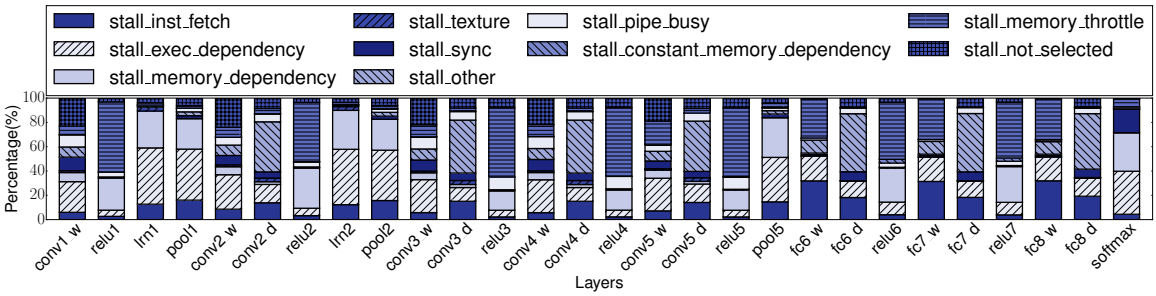


Figure 5.7: Stall breakdown for AlexNet running backward propagation on the K40.

5.2.2 Characteristics Analysis of Layers

Next, we delve a step deeper into the microarchitectural details that can explain the difference in terms of performance observed across different layers. First, we highlight the reasons for stalls during kernel execution on the baseline K40 platform in order to understand the characteristics of each layer. Figure 5.7 shows a breakdown of the stalls in each layer of AlexNet while running on the K40.

Based on the stall categories chosen, we can identify the major bottlenecks present in each layer, identifying the two largest contributors. We select conv2_w, relu2, lrn2, pool2, fc6_w, and softmax to represent convolution, activation, LRN, pooling, fully-connected, and softmax layer, respectively. As indicated in the figure, the two dominating stall categories for conv2_w are *stall_exec_dependency* and *stall_not_selected*. The former indicates the intrinsic program characteristics of this layer, meaning that there are many dependencies during instruction execution within a warp. The latter implies the warp is not selected to run since the scheduler selects competing warps. In other words, the SMs are always busy when warps are scheduled. Therefore, the performance of conv2_w is mainly bound by compute resources. The two major stall reasons for relu2 are *stall_memory_throttle* and *stall_memory_dependency*. The sources of these stalls are obvious. The former is caused by memory bottlenecks, and the latter is due to program characteristics related to data dependencies when executing memory loads and stores. Hence, the relu2 layer is memory-bound. Due to the dominant reasons for stalls in lrn2, pool2, and softmax as indicated in the figure, they are all compute and memory bound. fc6_w is somewhat special in that it is partially bounded by memory and partially bounded by instruction fetch. Even though there is little public documentation describing how instruction fetch works on Nvidia GPUs, we believe it should be related to the performance of the warp scheduler, which will be explained later when we look at the results of the GTX1080.

Given the characteristics of the representative layers/ primitives, it is expected that performance gains will vary on differnt hardware. Generally, a higher core clock rate and more SMs should benefit compute-intensive applications more, while increasing the memory clock rate only creates limited benefits. Since data load/write performance is not only dependent on the memory clock rates, but also on the demands and bandwidth of the different memory components in the memory hierarchy, the improved performance is due to using a faster memory clock rate. But this is only true for memory-bound applications. In contrast, higher processor clock rates and larger number of SMs will have a direct impact on the FLOP rate. Hence, this explains variations in performance gains as we migrate AlexNet from the K40 to the GTX1080 (which has more SMs and higher processor and memory speeds).

In Figure 5.8 we present the breakdown of stalls when running AlexNet on the GTX1080. We can see how the distribution of stalls change when running with a higher clock rate and with a larger number of SMs. One interesting change is that *stall_memory_throttle* have been eliminated in all layers due the fact that the GTX1080 provides a faster data-path for moving data between cores and memory. For the relu2 layer, the major reason for stalls is tied to program characteristics that are highly dependent on memory operations. Increasing the memory frequency should lead to limited performance benefits, given that relu2 is similar to a streaming application with little temporal locality. The breakdown of stalls in conv2_w does not change much. Given that we see that the scheduler is choosing to run other warps, this layer still has headroom to improve if a higher GPU core clock rate is used, or SMs are added. The lrn2, pool2, and softmax layers become memory-bound on GTX1080, because the compute performance of the GTX1080 over the K40 has improved more versus the memory performance of the two systems. The fc6_w layer is both memory and compute bound. Since we see the same warp scheduling issues we encountered in the conv2_w layer, there is headroom to improve performance for fc6_w. The new scheduler design in the Pascal architecture significantly alleviates the problems experienced with instruction fetching on the K40, so the new scheduler is able to handle more warps [158].

Finally, we report ALU utilization for both platforms, considering that this metric reflects, to a greater extent, how well the hardware can be exploited. Figure 5.9 shows the ALU utilization while running AlexNet. As shown in the figure, almost all of the layers involved in linear transformations have higher ALU utilization levels on the GTX1080. There is one case in lrn1, where the utilization level on GTX1080 is lower. This is because the LRN layer becomes memory-bound on the GTX1080, due to the increased computing capability, requiring more data to be accessed, and results in more processing core idle time.

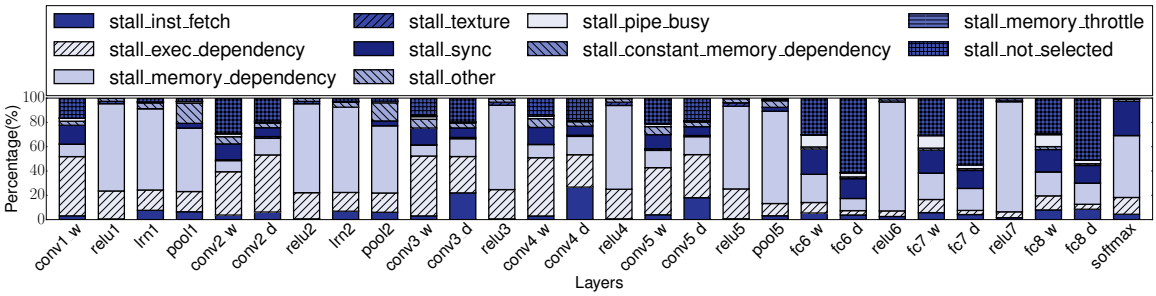


Figure 5.8: Stall breakdown for AlexNet, running backward propagation on the GTX1080.

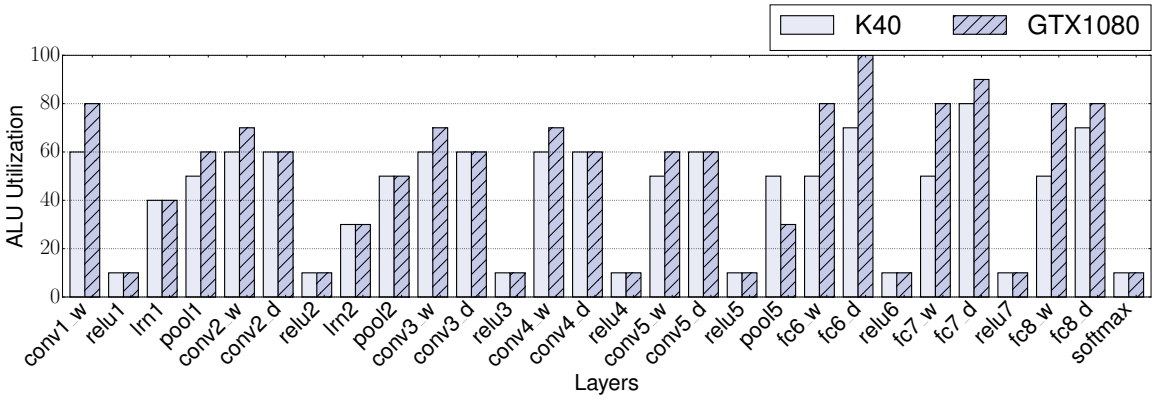


Figure 5.9: Compute unit utilization levels.

5.2.3 Memory Access Behavior

In this section, we focus on characterizing the memory behavior of each layer in AlexNet. Generally, in a discrete GPU, other than the main memory, the other critical memory component is the cache. The design of cache takes advantage of the locality present in applications, both in time and space, reducing the latency between instruction processing and memory access. In the GPU models we use, there are three different types of cache working together to support the streaming multiprocessors: i) an L2 cache, ii) a texture cache, and iii) an L1 cache, as shown in Figure 2.6. Additionally, there is also a fast on-chip scratch-pad memory, which is treated as GPU shared memory, for the programmer to use to achieve better performance. Depending on the memory space specified by the CUDA programmer, the processor will initially request the data from either the L1, shared memory, or the texture cache. If not present in any of the three locations, the data will be requested from higher levels in the memory hierarchy.

Given that the K40 and GTX1080 have different on-chip memory arrangements, as ob-

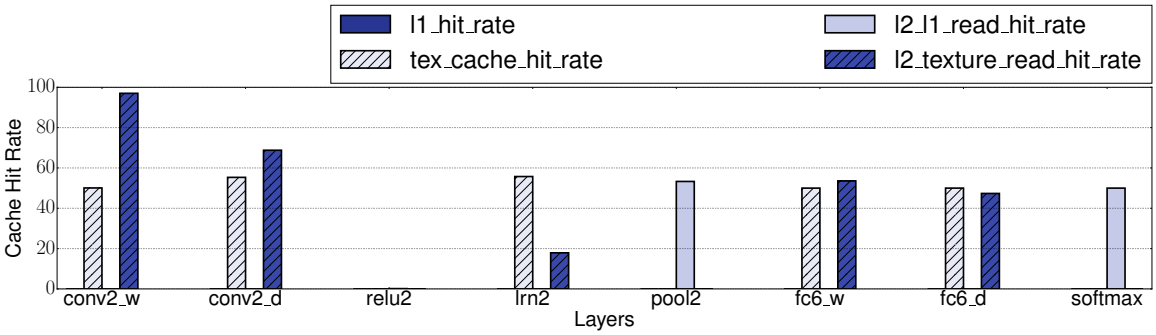


Figure 5.10: Cache hit rate for backward propagation of selected layers on the K40.

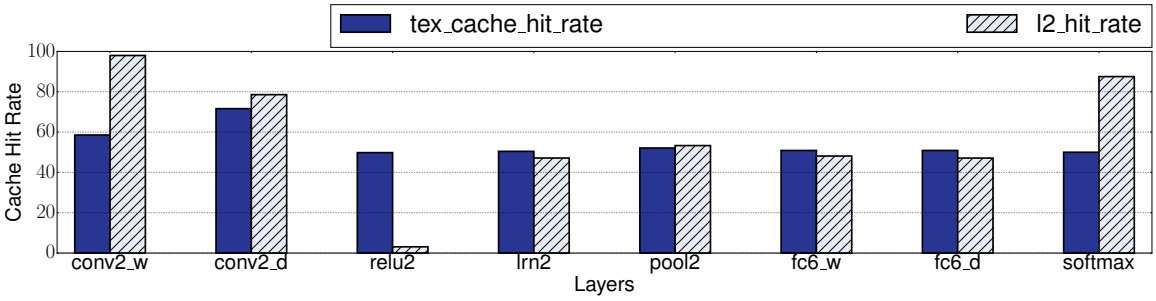


Figure 5.11: Cache hit rate for backward propagation of selected layers on the GTX1080.

served in Table 4.2, we showcase the cache hit rate on both platforms. In this section, we only present a subset of layers in backward propagation, using a batch size of 128 for simplicity, because the layers of the same type have very similar characteristics, as discussed earlier in our stall analysis and in our utilization evaluation. Likewise, we select conv2_w/d, relu2, lrn2, pool2, fc6_w/d, and softmax to represent convolution, activation, LRN, pooling, fully-connected, and softmax layer, respectively. We also only present backward propagation with a batch size of 128 for the same reasons as in our earlier discussion. Figures 5.10 and 5.11 present the cache hit rates of all caching components.

From the cache hit rates shown for the K40 in Figure 5.10, we notice that layers of the linear data transformation make good use of the texture cache, because both the convolution and fully-connected layers exhibit a high degree of spatial locality. The texture cache is designed i to take advantage of spatial locality. The L1 cache is a bit too small to handle this data. As a result, we see no L1 activities. In terms of the L2 hit rate, we can see cache accesses exist in almost every layer except the activation layers. This is caused by the element-wise operations present, so the activation layer acts as a streaming application with no temporal locality. Likewise, the texture cache is heavily utilized as well on the GTX1080. Even the activation layer, which has no temporal locality, makes

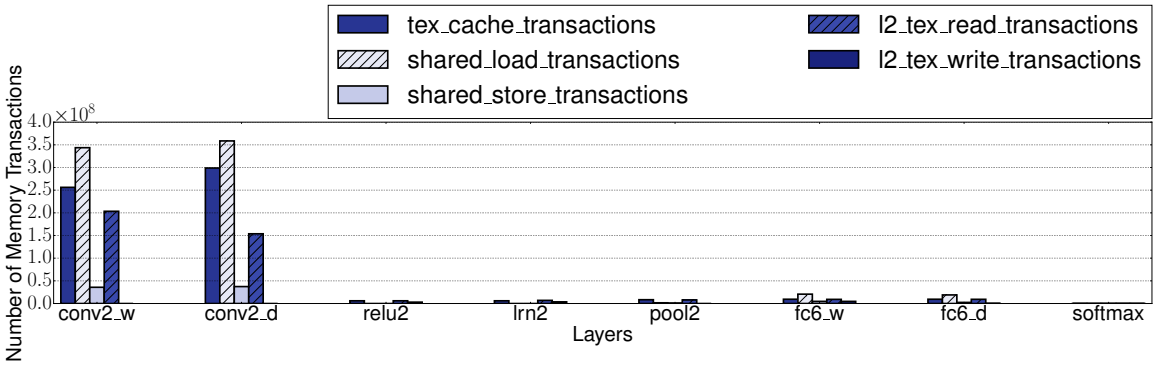
use of the texture cache to exploit spatial locality. To provide a fair comparison, we enable the L1 cache on the GTX1080 by toggling the corresponding compiler flag [66]. However, the L1 cache is unified with the texture cache, so it is difficult to observe any L1 cache activity merely through the texture cache hit rate. To address this issue, we compare the texture cache hit rate between the two cases where the L1 cache is enabled and then disabled. The results show no change in the texture cache hit rates, meaning that there is basically no L1 activity.

Next, we analyze the number of memory transactions and memory throughput for each level within the GPU memory hierarchy. Besides the improvements in memory throughput at every level of memory (thanks to the increased clock rate), we also notice an increase in the number of memory transactions handled by the memory components on the GTX1080, specifically in shared memory and L2 cache accesses. In contrast, the number of memory transactions issued to the DRAM is reduced significantly. This means that the larger L2 cache allows the kernels to better exploit locality, storing data closer to the processor and reducing DRAM request. Similarly, a larger shared memory provides more opportunities to store data structures that will be re-used frequently.

Although there are many differences in memory performance between the two GPUs, the trend in these metrics for the two platforms is still very similar in most cases. To provide perspectives from both the processor side and the memory side, we split the metrics into two parts, each of which represents the memory components closer to the processor or closer to the DRAM, respectively. Figures 5.12- 5.13 show the number of memory transactions and the memory throughput in various memory components. Note that we only present results from the GTX1080 because the trends are very similar.

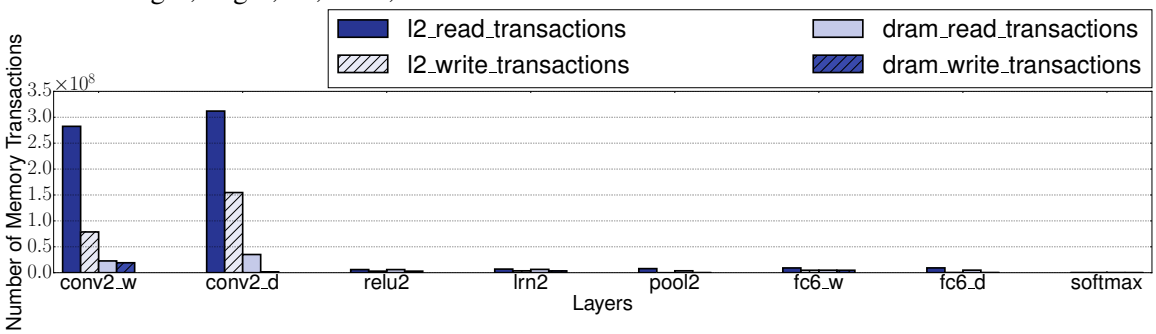
From Figure 5.12a, we can see that the linear data transformation layers rely heavily on shared memory and the texture cache. As indicated in the cache hit rate figures, both the convolution and fully-connected layers possess high temporal and spatial locality, given that data accessed within a region is repeatedly accessed. As a result, there are a large number of memory transactions issued to these two memory levels, especially read requests. This means that some shared memory data is heavily reused during the computation. On the contrary, for other layers, the utilization of shared memory and texture cache is very limited. Even for the pooling and LRN layers, the data reuse rate is very low. Figure 5.12b supports the previous statement. For the other layers, including pooling, LRN, activation, and softmax, the number of memory transactions does not vary significantly across the memory hierarchy.

With regards to memory throughput, In Figure 5.13a, we can see that shared memory throughput was almost 4x higher than on the texture cache, even though the number of memory



(a)

between images, e. g. , , , etc.



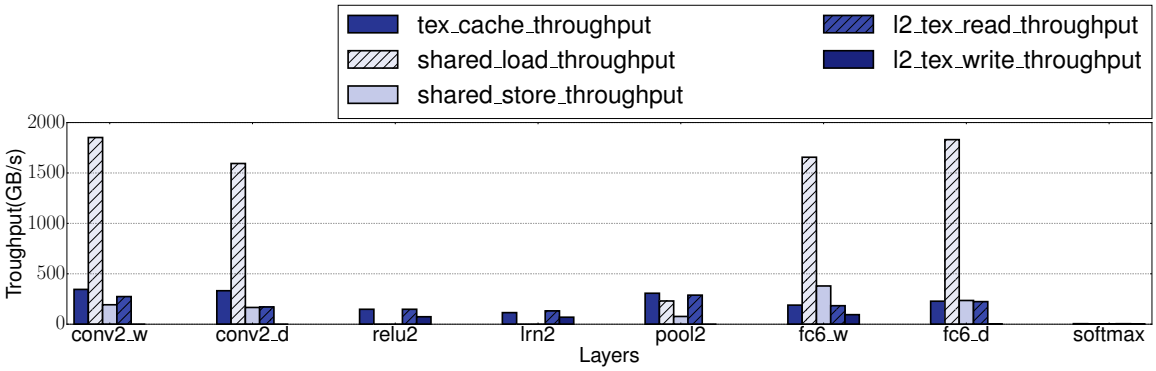
(b)

Figure 5.12: (a) Number of memory transactions in memory components closer to processor. (b) Number of memory transactions in memory components closer to DRAM.

transactions in these two components is similar for the convolution layers. This suggests that shared memory has much higher bandwidth than the texture cache. For instance, shared memory usually takes 38 cycles to read, while the texture cache takes 436-443 cycles [159]. The latest hardware has shortened the performance gap between those two, but the gap is still large. Thus, the bandwidth of the texture cache is a limiting factor for the convolution layers.

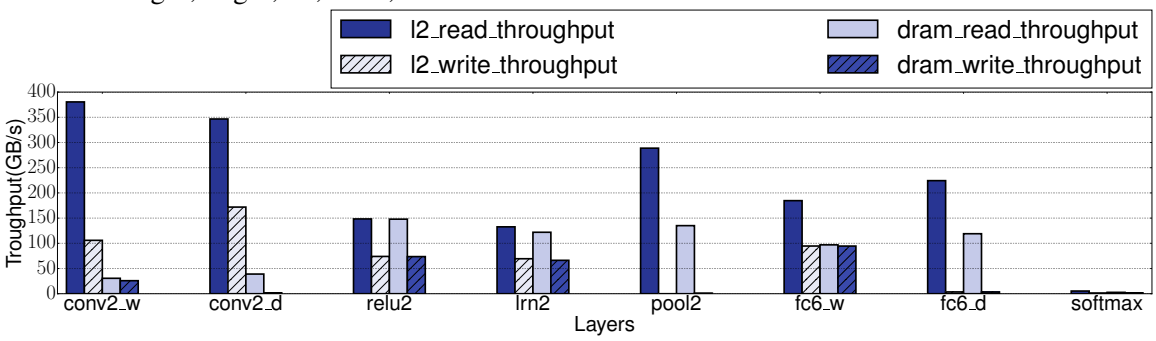
Note that increasing the bandwidth of texture memory without taking other associated memory components into account, could result in limited benefits. For example, if the texture cache becomes much faster than the L2 cache, the L2 cache will become the bottleneck. Increasing the bandwidth of the texture cache further would not benefit performance.

From Figure 5.13b, we can see that the throughput of DRAM in the activation layers is higher than that for the other layers. This is because the memory access patterns in the activation



(a)

between images, e. g. , , , etc.



(b)

Figure 5.13: (a) Memory throughput of memory components closer to processor. (b) Memory throughput of memory components closer to DRAM.

layers are more regular, meaning that memory requests can be coalesced, resulting in a better ratio between the size of useful data to the number of memory transactions. Given that throughput is computed using the size of the requested data, divided by the time between the first and last memory transaction, a higher ratio leads to higher throughput.

We evaluate the utilization of the memory hierarchy in Figure 5.14. We show how each layer utilizes individual memory levels in the hierarchy. We find that the convolution layers can leverage shared memory and the texture cache, while activation layers utilize the DRAM heavily during the execution.

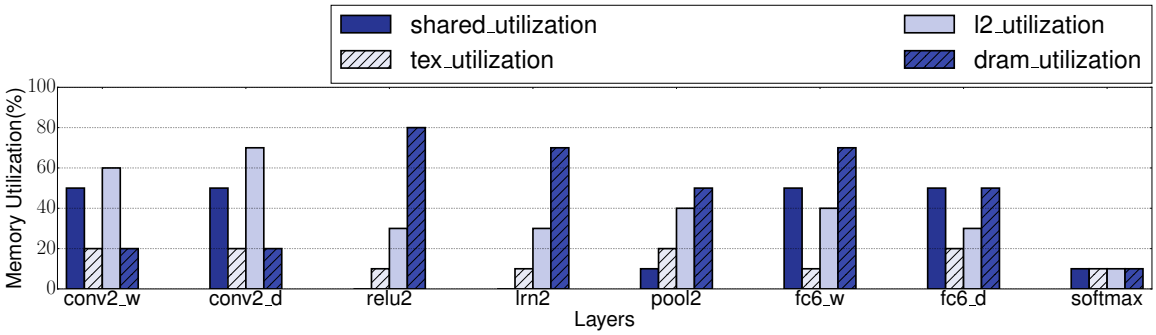


Figure 5.14: Memory components utilization of selected layers on the GTX1080.

5.2.4 Potential Optimization

From our analysis, we propose a number of design changes that can benefit CNN execution on GPUs, especially the GTX1080. First, we begin with the major bottleneck which are present in the convolution layer, as seen in Figures 5.3 and 5.4. Stalls during convolution are due both to intrinsic program characteristics and the limits of the hardware (even on the GTX1080). A simple solution is to add more SMs to the GPU. Increasing the DRAM bandwidth on the GTX1080 will not benefit the CNN throughput very much. Instead, if we increase the bandwidth of the texture cache, we should see much better performance. As discussed earlier, a significant number of memory transactions occur in the texture cache, but given its meager bandwidth, we see bottlenecks. Thus, we need to reduce the read latency of the texture cache.

Next, we find that L1 cache is essentially unused in most of the layers. The main reason is that the L1 cache is too small to hold data that has a strided access pattern. Based on this observation, we can enable L1 cache bypassing [160] for selected layers to avoid unnecessary data requests to the L1 cache. When we re-run our application with the L1 cache disabled for both reads and writes, we observe a speedup in some layers for both forward and backward propagation. From these results, we find that a single layer used in backward propagation (i.e., calculating the convolution layer weights) can achieve a 6.2% speedup on the GTX1080. However, this approach is limited in terms of achieving better overall application throughput. One issue is that some layers exhibit temporal locality, so L1 cache bypassing needs to be applied selectively in these layers. If we focus on optimizations that only benefits a subset of layers, the overall performance gains will be limited.

Another optimization we explore is to apply kernel fusion [161] for the linear data transformation layers and the non-linear activation layers. As observed in the results from the utilization breakdown, the activation layers place little pressure on compute resources due to their simple,

element-wise, operations. The idea is to combine linear data transformation and non-linearity. By doing so, we can eliminate all activation layers in the neural network model, leading to a significant reduction in the number of memory transactions, with only a small amount of added work performed in the linear data transformation layers. Although the runtime of the activation layers is insignificant, we can still save the time spent on running the driver and kernel launch, improving power efficiency as well. Given that cuDNN is not an open-source library, we are not able to further explore kernel fusion opportunities. So in order to evaluate the potential benefits of kernel fusion, we directly removed activation layers, assuming that the additional computation in the linear data transformation layers can be ignored. In this experiment, we measure the overall runtime, not just the kernel execution time. We are able to achieve a speedup of 4% on average. As the activation layers only take 3% of the overall execution time, we save approximately 1% the time spent on driver and kernel launch.

5.3 Discussion

CNNs are quickly becoming very important applications in a number of application domains. CNN computations have a distinct characteristics both in computing and memory requirements. Given the diversity of CNN applications, exploring characteristics of the basic primitives in a CNN is a pre-requisite when attempting to accelerate this class of applications. Some researchers have explored using custom hardware (i.e., ASIC and FPGA) for application-specific solutions [43, 44, 57, 45, 142]. However, the GPU is still a preferred accelerator for high-performance training, given that it provides a much simpler programming interface and larger memory space for storing large amounts of training data [64, 141, 86]. Our evaluation in this chapter focused on the microarchitectural demands associated with CNNs when mapped to two different NVIDIA GPUs. We evaluated how the same workload scaled on different platforms. We also analyzed microarchitectural metrics across different layers, considering the pressure placed on both compute units and memory components. We also proposed optimizations based on the observed bottlenecks and insights.

As presented in previous studies, there have been a number of optimizations proposed to maximize the performance of DNNs on GPUs. Most of these approaches focus on improving the memory efficiency [122], exploiting the data reuse pattern [123], or designing efficient GPU kernels [124]. However, the optimizations to increase parallelism and memory efficiency are reaching upper bound on performance [162]. Our characterization results suggest the same, i.e., proposing more efficient optimizations is challenging. Consequently, researchers are pursuing algorithmic

changes to DNNs to inspire novel methods to achieve further performance gains. Of the many approaches pursued, two have attracted our attention: 1) sparsity exploitation [43, 57, 138, 137] and 2) model compression [163, 90, 133, 164]. Both sparsity exploitation and model compression essentially reduce computation and memory accesses, potentially enabling novel hardware design and optimizations. In chapters 6 and 7, we present our work considering sparsity exploitation and model compression, respectively.

Chapter 6

Spartan: A Sparsity-Adaptive Framework to Accelerate DNN

The dominant computations used during DNN training are matrix-based operations (General Matrix Multiplication or GEMM). Spurred on by the emergence of high performance accelerators that are able to perform billions of matrix-based operations efficiently, training a large-scale DNN has become a reality. The time to train a DNN has been greatly reduced when using GPUs [155], thus enabling this class of algorithms to reach unprecedented popularity.

To support high-performance training of DNNs with large datasets (e.g. ImageNet [15]), GPU vendors started to design high-end platforms incorporating state-of-the-art GPUs and high bandwidth interconnects. The latest DGX-A100 system from NVIDIA can achieve 5 petaFLOPS with 8 NVIDIA A100 Tensor Core GPUs interconnected by Mellanox [165]. The latest AMD's machine learning specific MI-50 accelerator can achieve up to 26.5 TFLOPS FP16 and 13.3 TFLOPS FP32 [166].

In spite having more compute and memory resources added in each new generation of GPU, vendors are not kept pace with the requirements of DNN training (e.g., training Resnet-200 on the ImageNet dataset takes 3 weeks on 8 GPUs [167]). As a result, The computer architecture community has explored methods to improve execution efficiency and memory usage when processing DNNs. Of the many approaches pursued, leveraging weight/activation sparsity has attracted a lot of attention [43, 44, 64, 141, 45, 168]. Prior studies have explored exploiting sparsity to accelerate DNN computations during both training and inference on customized platforms (e.g., FPGAs and ASICs) [43, 44, 45, 142]. For inference, the major source of sparsity occurs after applying weight sparsification and

quantization [162]. In contrast to inference, for training, sparsity is produced by the ReLU activation function during both forward and backward propagation, and in the Max Pooling layer during backward propagation [64]. A recent study found that there can be as much as 90% activation sparsity in AlexNet [169].

Currently, leveraging sparsity for accelerating training mainly targets customized platforms [142]. Exploiting sparsity during training on a GPU has been largely unexplored, even though GPUs still serve as the primary platform for training large-scale DNNs [167]. GPUs can effectively accelerate both dense [156, 170] and sparse [61, 110] matrix operations. Given the large number of zero values detected during training, employing sparse matrix operations should be able to further reduce training time on a GPU. We focus on leveraging activation sparsity for the following reasons: 1) we observe limited weight sparsity (less than 1%) during DNN training, and 2) we do not explore weight sparsification and quantization methods in order to keep training lossless. We focus on accelerating convolutional layers, as they are the most compute-intensive layers in a DNN and take over 90% of the overall execution time [58].

DNN training presents many challenges when attempting to leverage activation sparsity. The training of a DNN is an iterative and dynamic process. The content of the activation maps keeps changing due to the randomly selected inputs (e.g., a batch of images) throughout the training. Therefore, efficiently leveraging activation sparsity during training presents the following challenges: i) tracking and profiling data sparsity introduces high overhead due to the data movement between the CPU and the GPU; ii) the contents of activations change dynamically throughout every iteration, requiring low-overhead dynamic sparse format conversion. The conversion overhead of current popular sparse matrix formats is too high for dynamic conversion, because: 1) the generation of the indexing information is inherently a serial process, and 2) multiple data structures are needed, involving many write operations. As a consequence, the overhead of dense-sparse matrix format conversion can outweigh the benefits of using sparse matrix operations.

In this chapter, we present *Spartan*, a sparsity-adaptive framework to accelerate training of DNNs on a GPU. We first characterize sparsity patterns present in activations used in DNN training. We also consider the current state-of-the-art approaches for managing sparse matrices. Then, we highlight the challenges of leveraging the sparse data while trying to accelerate DNN training. Based on key observations from our characterization results, we propose the Spartan framework to address these challenges.

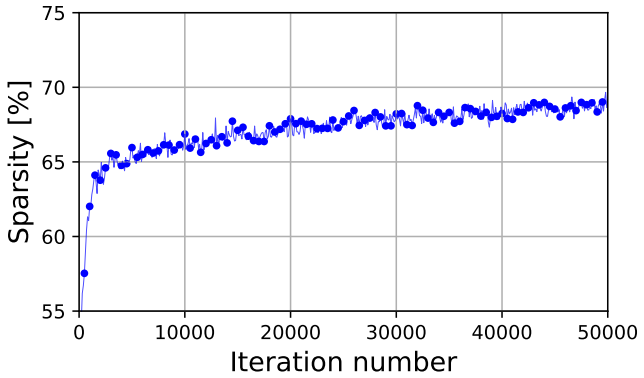


Figure 6.1: The sparsity trend from one activation map of CifarNet.

6.1 Motivation For Exploiting Sparsity During DNN Training

Next, we present our characterization of activation sparsity observed during DNN training. We also review the performance associated with state-of-the-art sparse matrix multiplication solutions. Given the high sparsity levels (up to 80%) observed during the training process, and the inherent inefficiencies of existing sparse matrix multiplication solutions [109, 110, 61], we are motivated to develop our Spartan framework that incorporates both software and hardware features. We focus on characterizing activation sparsity for two reasons: 1) we observe limited weight sparsity during DNN training, and 2) we do not explore weight sparsification and quantization methods in order to keep training lossless.

6.1.1 Activation Sparsity During DNN Training

Popular DNN models [16, 17, 18, 19] employ linear layers that include convolutional and fully-connected layers, as well as non-linear layers that include max-pooling layers and ReLU activation functions. The ReLU activation function outputs zeros if the input is negative, creating sparsity. Also, the max-pooling layer performs a downsampling operation during forward propagation and an upsampling operation during backward propagation. The upsampling operation also generates a large number of zeros.

Figure 6.1 illustrates the sparsity trends that occur during generation of one activation by a ReLU activation function after the second convolutional layer of CifarNet [171], trained using the CIFAR-10 dataset. From the figure, we notice significant sparsity (around 70%), even though there is great diversity in the input values across training iterations. We observe similar trends across

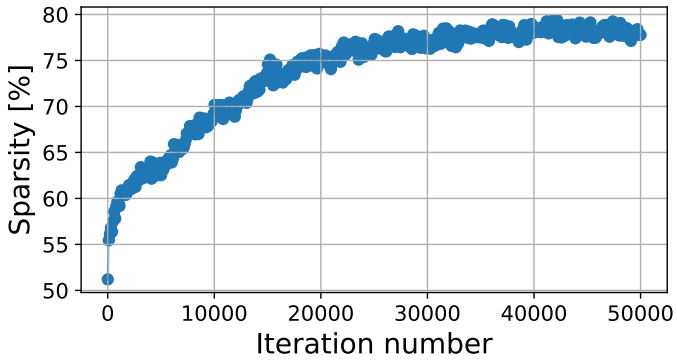


Figure 6.2: The sparsity trend from one activation map of AlexNet.

different variants of DNN models, such as AlexNet, VGGNet, and ResNet. Figure 6.2 show the sparsity trends of one activation of AlexNet trained with ImageNet data. We can summarize the sparsity patterns observed as follows:

1. The sparsity patterns and distribution of zeros change over time. This is because the training input changes on each iteration.

2. Activation maps from different layers contain different degrees of sparsity and sparsity trends over time.

3. The sparsity pattern exhibits negligible variance during short periods across consecutive training iterations.

4. The sparsity level increases gradually, then remains stable across many training iterations.

According to observations 1 and 2, we need an *appropriate sparse format conversion* to be *dynamically* convert data efficiently that is stored in a dense format to its sparse format, enabling effective use of sparse matrix operations on demand. In addition, observations 3 and 4 suggest that *exhaustive profiling of sparsity during every iteration is not needed*, motivating us to develop an efficient profiling mechanism.

6.1.2 Characterization of Sparse Matrix Operations

Both convolutional and fully-connected layers use General Matrix Multiplication (GEMM) as their primary computational kernel. The computation in the fully-connected layers can be directly represented by GEMM, while the computation in the convolutional layers can be a combination

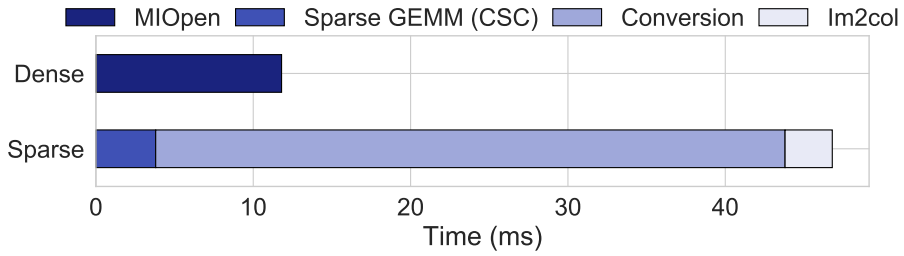


Figure 6.3: Execution time breakdown for convolution using clSparse and rocSparse, compared with MIOpen.

of an *im2col* operation (transforming high-dimensional activation/feature maps into a 2D matrix) and a GEMM operation [155]. Convolutional layers dominate the overall DNN training time. In particular, the convolutional layers alone can contribute to approximately 90% of the training time [58, 88]. Therefore, in this chapter we are focused on improving GEMM-based convolutional layer performance, given its dominance on training performance.

Next, we capture the execution time of a convolutional layer during the forward propagation operation, comparing the performance when using a popular Compressed Sparse Column (CSC) [172] and a dense format. We configure the convolutional layer using a filter size of $5 \times 5 \times 64 \times 64$ (CifarNet [171]). We evaluate the convolutional layer using a dense format with AMD's MIOpen, using both *im2col* and dense GEMM kernels [173]. When using a sparse format, we include the dense-to-sparse conversion (implementation provided by the clSparse [109] library) between the *im2col* and the sparse GEMM (using the implementation in the rocSparse library [110]).

The execution time breakdown, as shown in Figure 6.3, suggests that using a sparse matrix multiplication can significantly reduce the GEMM execution time. However, as the dense-to-sparse format conversion introduces large overhead (approximately 85% of the overall execution time), the overall execution time of the convolutional layers increases when using a sparse format. To exploit the sparsity present in DNN training, we need a new solution that can hide the dense-to-sparse conversion overhead and accelerate the convolutional layers.

6.2 Sparsity Monitor

In this section, we present our sparsity monitor design, providing efficient and flexible sparsity monitoring during DNN training. The sparsity monitor detects activation sparsity before each convolutional layer in a DNN model, determining when to leverage our sparse GEMM kernel

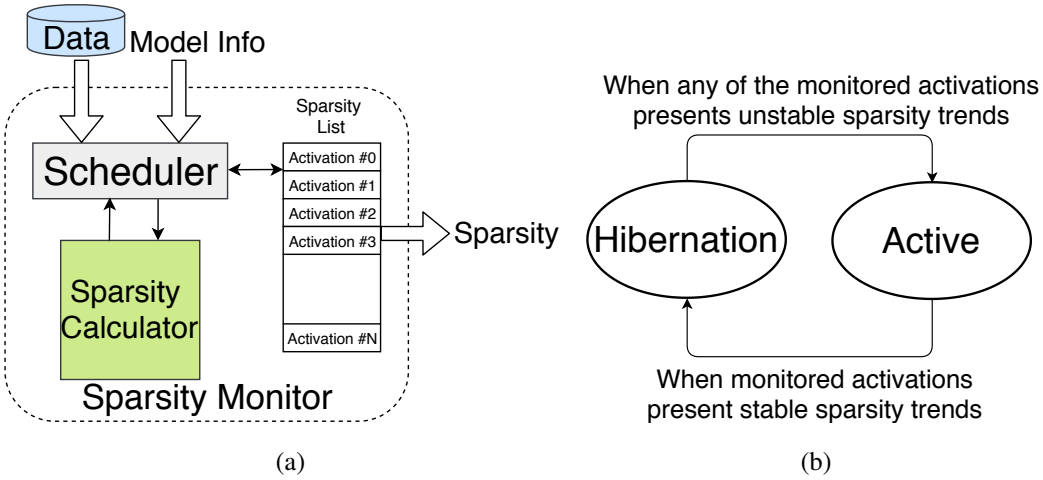


Figure 6.4: (a) The overview of the sparsity monitor. (b) State transition of Dynamic Monitoring Period Management.

acceleration dynamically, depending on the detected sparsity level. Running on the CPU, our monitor is designed to reduce the overhead of data movement between the CPU and GPU. Figure 6.4a presents an overview of the sparsity monitor, which consists of three components: 1) a scheduler, 2) a sparsity list, and 3) a sparsity calculator. The regular workflow of the sparsity monitor is as follows. First, the model information (i.e., the structure of the DNN model and a list containing which activation maps we select to monitor) is sent to the scheduler to initiate the monitoring process. Next, the scheduler determines when to profile the activation maps and enables the sparsity calculator to compute the degree of sparsity based on the selected activation maps (Data). Then the scheduler manages and updates the sparsity list that contains the sparsity for individual activation maps being monitored.

The scheduler manages and monitors each individual activation map by regulating two important parameters: 1) the monitoring period, and 2) the monitoring duration. The former determines the timing gap between two monitoring processes, while the latter indicates the length of the monitoring process. To avoid exhaustive monitoring, the monitoring duration should be smaller than the monitoring period. Once determined by the user, the monitoring duration remains the same across all monitoring processes. The monitoring period, on the other hand, can be dynamically changed to further reduce the profiling overhead. Specifically, we consider the importance of observations 2, 3 and 4 presented in Section 6.1.1. We propose multiple mechanisms to assist with periodic monitoring, dynamically adjusting the monitoring period. The mechanisms are: i) flexible

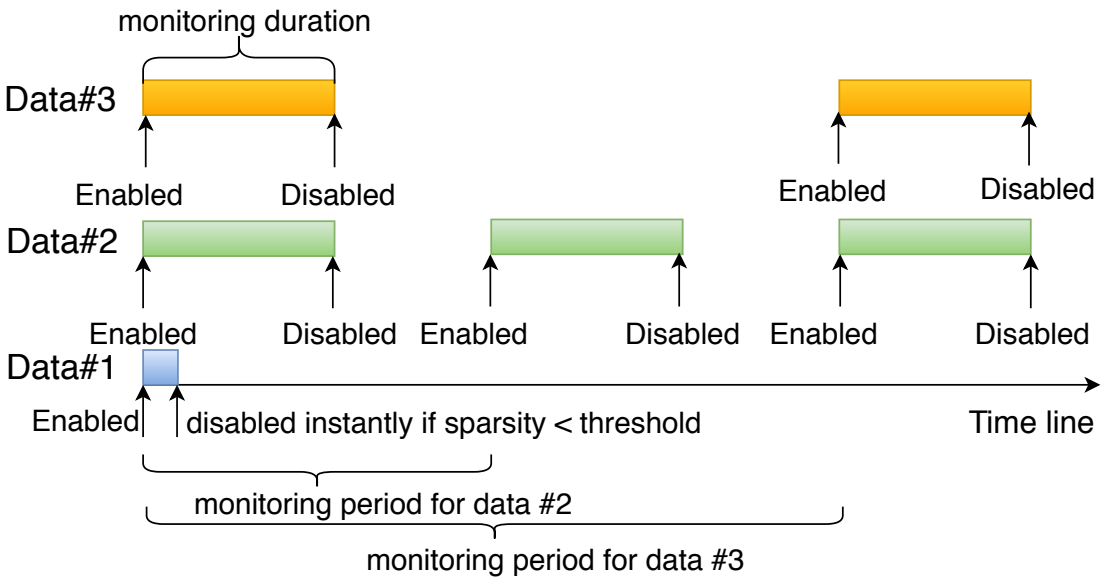


Figure 6.5: Three monitoring processes with flexible monitoring and fast monitoring disabled.

monitoring, ii) fast termination, and iii) dynamic management of the monitoring period management.

i) Flexible Monitoring provides a mechanism to regulate the monitoring period in a flexible manner. For each individual activation map, the monitoring process can have a different monitoring period. As per observation 2, activation maps from different layers present different levels of sparsity and the sparsity associated with these activation maps may change over time. Therefore, flexible monitoring can be more efficient in managing the monitoring process than using only a single monitoring period. Figure 6.5 shows two examples of a flexible monitoring process. In particular, the Data#2 (green monitoring process) and Data#3 (orange monitoring process) are two monitored activation maps that have different monitoring periods.

ii) Fast Termination stops the monitoring process if the detected sparsity level is below a user-defined threshold. By doing this, the monitoring duration is reduced to avoid useless monitoring. Figure 6.5 also shows an example of fast termination where the monitoring process stops for Data#1 (blue monitoring process) immediately after a low sparsity level is detected.

iii) Dynamic Monitoring Period Management is a key mechanism for reducing data movement overhead, tuning the monitoring period dynamically according to the sparsity variance observed throughout many training iterations. Dynamic Monitoring Period Management maintains a two-state finite state machine: 1) *Active* and 2) *Hibernate*. We introduce these two states to handle two possible scenarios according to observation 4: 1) when sparsity changes gradually and 2) when

sparsity remains at a stable level. In the Active state, the monitoring period is shorter and can dynamically respond based on a Dynamic Period Adjustment Algorithm (DPAA), depending on the current sparsity trends. While in the Hibernate state, the monitoring period is longer and shared by all monitored activation maps for simplicity (further details below). The transitions between the two states are shown in Figure 6.4b. The conditions for transitions are regulated by Algorithm 2, a Sparsity Stability Detecting Algorithm (SSDA).

DPAA is enabled only in the Active state, adjusting the monitoring period of each individual activation map. The DPAA (Algorithm 1) can adjust the monitoring period by maintaining history and storing the most recently measured sparsity for every monitored activation map managed in the sparsity list. The algorithm first collects the measured sparsity and adds it to a history list, saving only the last few measured sparsity levels. Then it checks the difference between the most recent sparsity and the oldest in the history list. If the absolute difference is smaller than a threshold, we double the length of the monitoring period in the Active state. The process continues until the monitoring period is larger than the one used in the Hibernate state.

SSDA detects the stability level of sparsity, determining when to transition between the Active and Hibernate state. As described in Algorithm 2, in the Active state, the algorithm polls through the sparsity list in Figure 6.4a. If the sparsity levels of all selected monitored activation maps become stable, the state can transit to Hibernate, a state where all activation maps share the same monitoring period. While in the Hibernate state, if any of the monitored activation maps exhibit unstable sparsity trends, the state transits to Active, resetting the monitoring period for all monitored activation maps to the initial monitoring period.

6.3 ELLPACK-DIB Based GEMM

In this section, we explore a novel sparse format named *ELLPACK-DIB* and a tile-based GEMM algorithm designed to effectively exploit this format.

As discussed in Section 6.1, when using the CSC format, the dense-to-sparse conversion is costly in terms of execution time. Regular sparse formats, such as CSR/C [172] and COO [172], use multiple data structures to store the non-zero elements and indexing information. From our analysis of these formats, we have identified two major factors contributing to the dense-to-sparse conversion overhead. First, given a sparse matrix with size $M \times N$, storing the non-zero elements and calculating indexing information comes with a time complexity of $O(M \times N)$. This is because storing a non-zero element and calculating the associated indexing information depends on the location and index of

Algorithm 1 Dynamic Period Adjustment Algorithm (DPAA)

```
1:  Inputs: sparsity, history, max_length, threshold, active_period, hibernate_period
2:  Outputs: N/A
3:  if sizeof(history) < max_length then
4:                                                    ▷ Collecting the sparsity history
5:      idx = sizeof(history)
6:      history[idx] = sparsity
7:  else
8:                                                    ▷ Increase the monitoring period
9:      if abs(sparsity - history[0]) < threshold then
10:         active_period *= 2
11:     end if
12:     if active_period > hibernate_period then
13:         active_period = hibernate_period
14:     end if
15:     history.Update(sparsity)
16: end if
```

the previous non-zero element. This dependency leads to a serialized conversion process that lacks any parallelism. Second, we encounter a number of writes needed to update multiple data structures for the non-zero elements and associated indexing information. Note that writes are commonly expensive and should be avoided as much as possible.

We find that one variant of ELLPACK, ELLPACK-R [113], has a structure wherein conversion can be parallelized. ELLPACK-R in row-major order requires three data structures. The first one stores non-zero elements. The second one stores the column index. The third one stores the number of non-zero elements per row. The data structures for each row are completely independent of each other, meaning that the dense-to-sparse conversion can be parallelized, reducing the time complexity to $O(N)$.

Although the time complexity of the conversion is significantly reduced, there are still three data structures. To simplify the data structures, we propose ELLPACK-DIB (Data Index Bundling). Considering that the data structures for storing non-zero elements and column indices are exactly the same size in ELLPACK-R, ELLPACK-DIB bundles the data and the indices together, storing them in a single data entity. Since we only use ELLPACK-DIB for activation maps rather than weights,

Algorithm 2 Sparsity Stability Detecting Algorithm (SSDA)

```
1:  Inputs: state, sparsity_list, sparsity, history, max_length, threshold, active_period, hiber-
    nate_period
2:  Outputs: N/A
3:  if state == ACTIVE then
4:      hibernation_ready_count = 0
5:      for i = 1 to sizeof(sparsity_list) do
6:          if active_period == hibernate_period then
7:              hibernation_ready_count++
8:          end if
9:      end for
10:     if hibernation_ready_count == sizeof(sparsity_list) then
11:         state.transit(HIBERNATE)
12:     end if
13: end if
14: if state == HIBERNATE then
15:     if sizeof(history) < max_length then
16:         idx = sizeof(history)
17:         history[idx] = sparsity
18:     else
19:         if abs(sparsity - history[0]) >= threshold then
20:             active_period.reset(all)
21:             state.transit(ACTIVE)
22:             history.clear()
23:         end if
24:     end if
25:     history.Update(sparsity)
26: end if
```

we use a *column-major* ELLPACK-DIB format, as shown in Figure 6.6, producing an efficient data access pattern for sparse GEMM.

Figure 6.6 presents the ELLPACK-DIB format in detail. As indicated in the figure, we only need two data structures to store the non-zero elements bundled - the row index, and the number

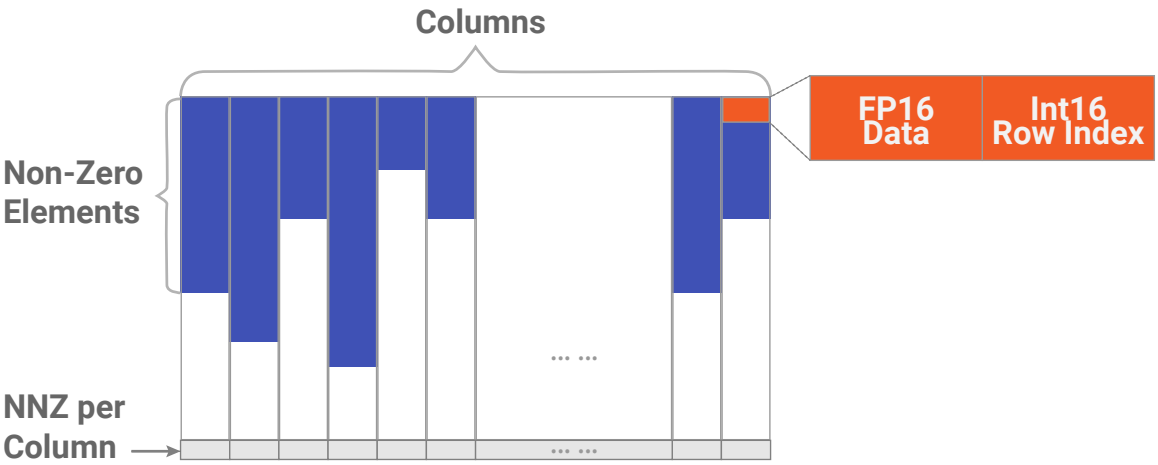


Figure 6.6: The format of ELLPACK-DIB.

of non-zero elements (NNZ) per column. We use IEEE half-precision floating point format (FP16) for the data and a 16-bit unsigned integer for the row index. We only use this format for converting activations during DNN training. The actual computation in the sparse GEMM (described below) still uses single precision (FP32) by converting FP16 data back to FP32. Therefore, the precision lost during conversion has little impact on the overall training performance [64, 174]. For the row index, a 16-bit unsigned integer representation is sufficient for all possible problem sizes in commonly-used DNN models. In particular, the largest convolutional layer in both ResNet and VGGNet has 4,608 elements ($3 \times 3 \times 512$) in one column after performing the *im2col* operation [19, 17].

We propose a tile-based GEMM algorithm using ELLPACK-DIB on GPUs. Algorithm 3 presents the kernel details. Note that we use some OpenCL kernel parameters, e.g., local_id and group_id [175]. The mapping functions (i.e., MappingRow, MappingCol, MappingTile, etc.), shown in the algorithm, are used to map the kernel parameters to a specific location for memory access.

Our algorithm involves three matrices: 1) A, a dense matrix for weights, 2) B, a sparse matrix for activations in ELLPACK-DIB format, and 3) C, a dense matrix which is the output. The parameters lda, ldb, ldc correspond to the leading dimension of matrices A, B, and C, respectively. Usually, the leading dimension is equal to the number of rows of the matrix. Each tile of matrix C is mapped to a workgroup [175], and each element of the tile is mapped to a workitem/thread [175]. The threads in one workgroup first load a tile of B in ELLPACK-DIB format, unpacking the data and row index accordingly. Then a tile of A is loaded, based on the row index. Once tiles of A and B are ready, multiplication and accumulation (MAC) are performed. This process is done iteratively

Algorithm 3 Tile-based GEMM algorithm Using ELLPACK-DIB (GPU kernel)

```
1:  Inputs: A, lda, B, ldb, ldc, nnz_col, tile_size
2:  Outputs: C
3:  row = MappingRow(group_id, local_id)
4:  col = MappingCol(group_id, local_id)
5:  sum = 0
6:  local_B[tile_size][tile_size]
7:  local_A[tile_size][tile_size]
8:  for i = 1 to nnz_col[col] with step tile_size: do
9:      idx = MappingTile(i, col, ldb)
10:     x = local_id / tile_size
11:     y = local_id mod tile_size
12:     local_B[x][y] = low2float(B[idx])
13:     row_idx = high2int(B[idx])
14:     local_A[x][y] = A[row + row_idx * lda]
15:     Synchronization
16:     for j = 1 to tile_size do
17:         b_idx = MappingLocalB(j, local_id)
18:         a_idx = MappingLocalA(j, local_id)
19:         sum += local_B[b_idx] * local_A[a_idx]
20:     end for
21: end for
22: C[row + col * ldc] = sum
```

until we reach the number of non-zero elements per column. To avoid load imbalance due to the varying number of non-zero elements, we zero-pad the last tile of B, thus avoiding potential branch divergence within a wavefront.

Figure 6.7 shows an example of this algorithm. In the example, we have four tiles in C, colored gray, blue, orange and yellow. These four tiles share the three tiles (named T0, T1 and T2) of matrix B in the ELLPACK-DIB format. Only three tiles are needed, as the non-zero elements only occupy three tiles in this example. To compute the gray tile, the data from matrix A (marked with gray to match the color of the gray tile in matrix C) is selected based on the row index loaded from tile T0, T1 and T2. The white area in A indicates the unselected data. We follow a similar process

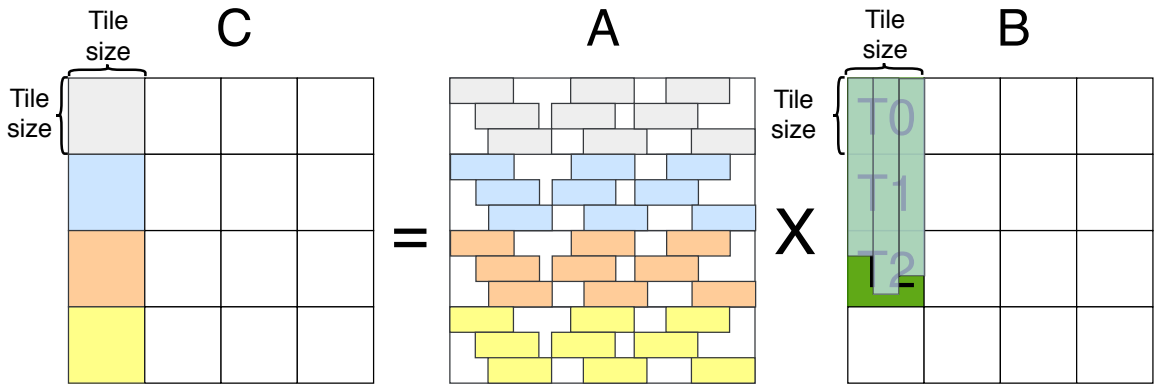


Figure 6.7: A tile-based GEMM using ELLPACK-DIB.

for the blue, orange and yellow tiles.

We evaluate the performance of dense-to-sparse conversion of ELLPACK-DIB and the customized GEMM algorithm using this format. Compared with CSC, the conversion time is reduced by over 10× and the time for GEMM is also reduced. Using the ELLPACK-DIB format, we can achieve a 19% speedup over MIOpen given a 90% sparsity, for the same problem size as indicated in Section 6.1. However, when we decrease the sparsity (e.g., 80%), the speedup is gone. As such, the benefits are very limited due to the conversion overhead. To address this overhead, we develop a hardware-based conversion mechanism, which is described in the next section.

6.4 Compaction Engine

In this section, we present the design of a novel hardware component named the *compaction engine*. The compaction engine is designed as a near-memory processing unit. The main purpose of it is to convert/compact sparse data to the ELLPACK-DIB format during kernel execution, hiding conversion overhead. The compaction engine consists of three major sub-components: the sparsity information block or SIB (Section 6.4.2), the compaction processor (Section 6.4.3), and the prefetch buffer (Section 6.4.4).

6.4.1 Overview

Figure 6.8a presents a logical view of a GPU equipped with a compaction engine. As shown in the figure, the compaction engine sits logically between the L2 cache and main memory, serving all banks of main memory/L2 and all Compute Units (CUs) of a GPU.

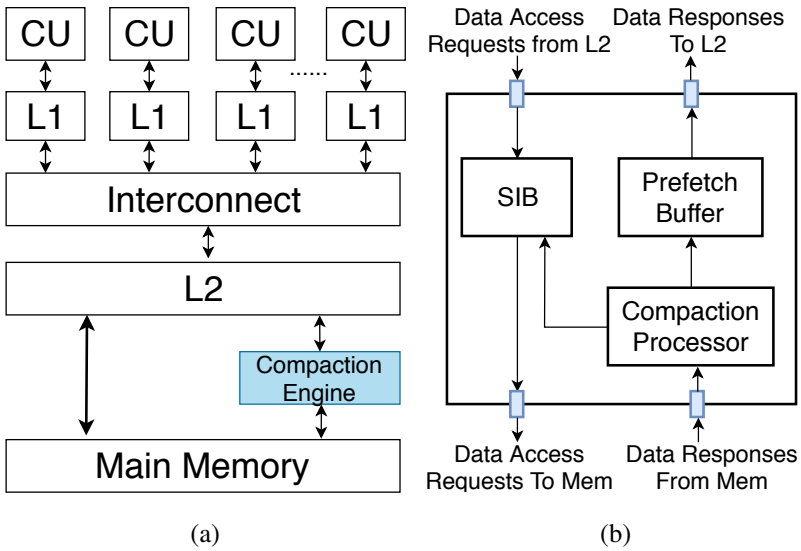


Figure 6.8: (a) GPU with a compaction engine. (b) Overview of the compaction engine.

The compaction engine provides compacted data in the ELLPACK-DIB format in a programmer-transparent manner. Figure 6.9 shows an example of handling a single memory request in our modified GPU with the compaction engine. Note that the compaction engine only serves data requests for reads, since only activations represented in matrix B during DNN training need compaction/conversion when running sparse GEMM. The reasons why we do not consider converting the activation maps through writes are twofold: 1) writes are costly, 64.3% slower than reads [176]. As such, converting the activation maps on writes leads to a performance degradation, and 2) dense GEMM is still used during training when the sparsity level is low. Spartan only enables sparse GEMM and the compaction engine after detecting significant levels of sparsity. From Figure 6.9, the steps to handle a memory request before the compaction engine are no different than those present in a regular cache system. The only difference is that memory requests issued from components above the compaction engine (i.e., CUs, L1s and L2) expect data in ELLPACK-DIB format. The compaction engine is responsible to respond to memory requests for loading sparse data, prefetching data stored in the original matrix format, compacting the data using the ELLPACK-DIB format and then returning them.

Prefetching overhead may lead to long latency when the requested compacted data requires a large amount of data from main memory. The worst-case scenario occurs when we need to load an entire column to service a single memory request. To avoid long-latency prefetching, we introduce a sparsity information block (SIB), a hardware component to store the profiled sparsity

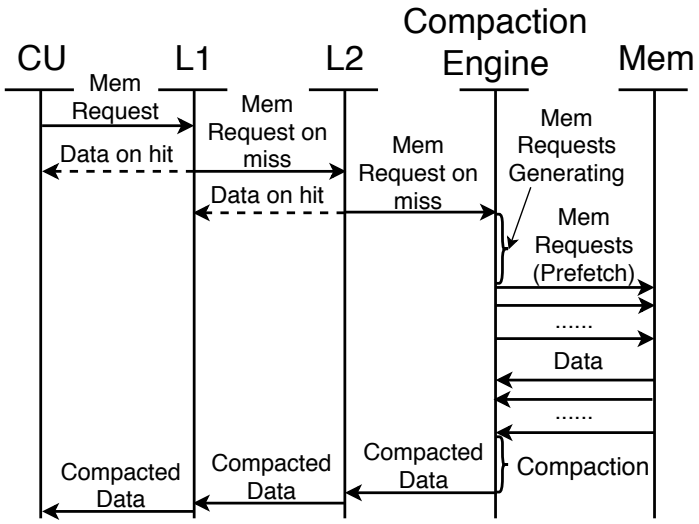


Figure 6.9: An example of handling a data read request.

information while performing prefetching. The primary sparsity information stored in the SIB is a series of bitmasks, where a value of "1" indicates a cache line with non-zero elements and a value of "0" corresponds to a cache line with only zeros. We describe the details of managing the SIB in Section 6.4.2. With the sparsity information stored in the SIB, the compaction engine can carry out a more intelligent and efficient way to issue prefetches, reducing prefetching latency. In addition to the SIB, we also add a compaction processor and a prefetch buffer. The former compacts the prefetched data into ELLPACK-DIB format and profiles the sparsity information at the same time. The latter stores the compacted data and streamlines the process of sending data to the L2 cache.

Figure 6.8b presents an overview of the compaction engine with the added components. To redirect data fetches of sparse data to use the compaction engine, we add an additional bit in the memory request, and extend an existing vector load instruction (FLAT_LOAD [177]) to indicate this special type of data load. Upon receiving a memory read request for the sparse activation maps (sparse memory request), the SIB issues memory transactions for prefetching, based on the data search algorithm we propose (further details in Section 6.4.2.1). When the data arrives from main memory, the compaction processor is activated to perform: 1) compaction, and 2) profiling. After that, we update the SIB contents using the profiled sparsity information and store the compacted data in the prefetch buffer. Once the data is ready for the corresponding request, the prefetch buffer will send it to the L2. We describe the details of the SIB, the compaction processor and the prefetch buffer in Sections 6.4.2, 6.4.3 and 6.4.4, respectively.

With the compaction engine, we no longer need a costly software-based dense-to-sparse conversion, and we no longer need an extra data structure for storing the number of non-zero elements per column, as shown in Algorithm 3. To achieve the latter, we add an additional bit in the data response to indicate whether or not the compacted data is the last one in the column. Instead of using an additional structure nnz_col, the program uses this additional bit, set by the compaction engine, to terminate.

6.4.2 Sparsity Information Block

The sparsity information block (SIB) consists of multiple entries for storing sparsity profiles for each column. Each SIB entry contains two types of sparsity information (further details in Section 6.4.2.2): 1) the sparsity bitmask, and 2) the cache line mapping information (CMI). Because the basic unit managed in the memory system is a cache line (i.e., 64 bytes or 16 single precision (FP32) elements) [177], the SIB also manages the sparsity information on a cache line basis.

The sparsity bitmask contains a series of bits, representing the sparsity pattern present in the uncompacted data in a column. For each bit, a value of 1 indicates a non-zero cache line which has non-zero elements, whereas a 0 value indicates a cache line with only zero elements. The position of the bits is relative to the position of the cache line in the associated column. For example, the first bit indicates the cache line starting at row number 0, the second bit indicates the cache line starting at row number 16, and so on.

The CMI records the mapping information between the compacted data and the uncompacted data (from main memory). Each mapping entry is associated with one compacted cache line (16 non-zero FP32 elements) and has two parameters: 1) the number of non-zero cache lines needed for filling 16 non-zero elements (NNC_needed), and 2) the offset indicating the last non-zero element in the last non-zero cache line. The CMI has multiple entries, each of which corresponds to a compacted cache line.

Figure 6.10 shows an example of setting the sparsity bitmask and CMI entries. In this example, there are 7 sparse cache lines (numbered #0 to #6) from main memory. Among them, only three are non-zero cache lines, shown in blue (#1, #4 and #6). The sparsity bitmask can be updated based on this layout. These 7 sparse cache lines can be compacted into 2 compacted cache lines, shown in green (#0 and #1). Compacted cache line #0 has data from sparse cache lines #1 and #4. The last element can be found in cache line #4, with an offset of 7. Thus, in CMI #0, the NNC_needed value is set to 2 and the offset is set to 7. The compacted cache line #1 holds data

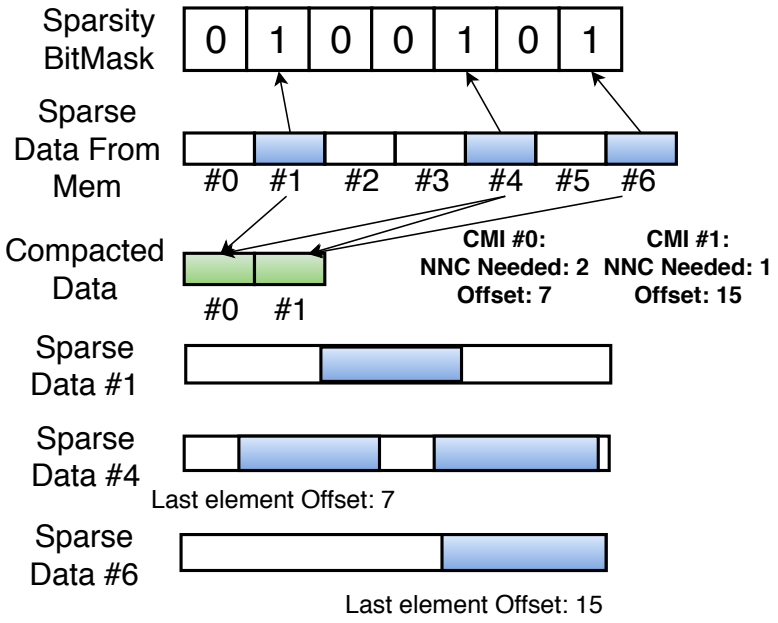


Figure 6.10: An example of updating the sparsity bitmask and cache line mapping information.

from cache lines #4 and #6. Note that this compacted cache line also requires two sparse cache lines. However, the previous cache line (#4) is prefetched before setting CMI #1. As a result, we set the NNC_needed to 1 and the offset to 15.

6.4.2.1 Data Search Algorithm

In order to avoid long-latency prefetches, we propose a data search algorithm with three search modes that leverages the sparsity information of the SIB (Algorithms 4 and 5). Guided by the sparsity bitmask and the CMI contents, the algorithm is able to issue memory transactions in a more intelligent manner, avoiding cache line accesses that do not contain the requested data.

The three search modes of the algorithm are as follows: 1) accurate search, 2) prudent search and 3) hasty search. Each of these modes corresponds to one potential scenario. For example, accurate search is used when the SIB holds the target CMI entry. The data search algorithm can find the locations of sparse data in main memory based on the information stored in the CMI entry. The prudent search is used when no CMI entry can be found, but a sparsity bitmask exists. This scenario can occur when a CMI entry is replaced by a newer entry. In this scenario, memory transactions are issued based only on the bitmask. The hasty search mode is used when neither a sparsity bitmask nor a CMI entry exists, i.e., the application is in the warm-up phase.

Function Name	Description
FindFirstNonZeroBit	Find the location of the first non-zero bit.
FindNextBitLocation	Find the location of a bit which is N non-zero bits away from a given bit location. N is an integer number.
FindNextNonZeroBit	Find the location of a bit which is the next non-zero bit from a given bit location.

Table 6.1: Descriptions of functions used in the data search algorithm.

Algorithm 4 Data Searching Algorithm (Accurate Search)

```
1:  Inputs: bitmask, CMI, address
2:  Outputs: mem_trans[ ]
3:  entry = GetCMIEntry(CMI, address)
4:  bit_location = FindFirstNonZeroBit(bitmask)
5:  for i = 0 to entry.idx-1 do
6:      bit_location = FindNextBitLocation(bitmask, bit_location, CMI[i].NNCNeeded)
7:  end for
8:  for j = i to entry.NNCNeeded do
9:      mem_tran = AddrCalc(bit_location)
10:     mem_trans.append(mem_tran)
11:     bit_location = FindNextNonZeroBit(bitmask, bit_location)
12: end for
```

Table 6.1 provides descriptions of the bitmask functions used in Algorithms 4 and 5. The major operation needed for implementing these functions is a bit shift.

Our search implementation accurately locates the bit locations, identifying where the memory transactions should be issued, based on the cache line mapping information. The algorithm loops through all of its previous CMI entries, using the parameter NNC_needed to arrive at the desired

Algorithm 5 Data Search Algorithms (Prudent Search and Hasty Search)

```
1:  Inputs: bitmask, prefetch_length, address, search_mode
2:  Outputs: mem_trans[ ]
3:  if search_mode == Prudent then
4:      bit_location = FindFirstNonZeroBit(bitmask)
5:  end if
6:  if search_mode == Hasty then
7:      bit_location = 0
8:  end if
9:  for j = i to prefetch_length do
10:     mem_tran = AddrCalc(bit_location)
11:     mem_trans.append(mem_tran)
12:     if search_mode == Prudent then
13:         bit_location = FindNextNonZeroBit(bitmask, bit_location)
14:     end if
15:     if search_mode == Hasty then
16:         bit_location++
17:     end if
18: end for
```

location. Then it issues the memory transactions based on its own NNC_needed value. Prudent search first issues memory transactions from the location of the first non-zero bit in the bitmask. Then it traverses through the bitmask, issuing memory transactions only based on the non-zero bits. The hasty search issues memory transactions in a consecutive manner from the first bit of the bitmask.

6.4.2.2 SIB Management

Considering that the size of the compacted data in a column can vary with different column sizes, as well as the sparsity level, we adopt an approach to dynamically determine the entry size and allocate space for the sparsity bitmask and CMI. Figure 6.11 shows the contents of a SIB entry.

From the figure, the SIB entry is composed of three parts: 1) metadata, 2) a sparsity bitmask and 3) cache line mapping information. The metadata has fixed 6 bytes used for storing meta-information, including column number, current bit count in the bit mask and current number of valid CMI entries. We define K as the entry size in bytes, N the sparsity bitmask size in bytes,

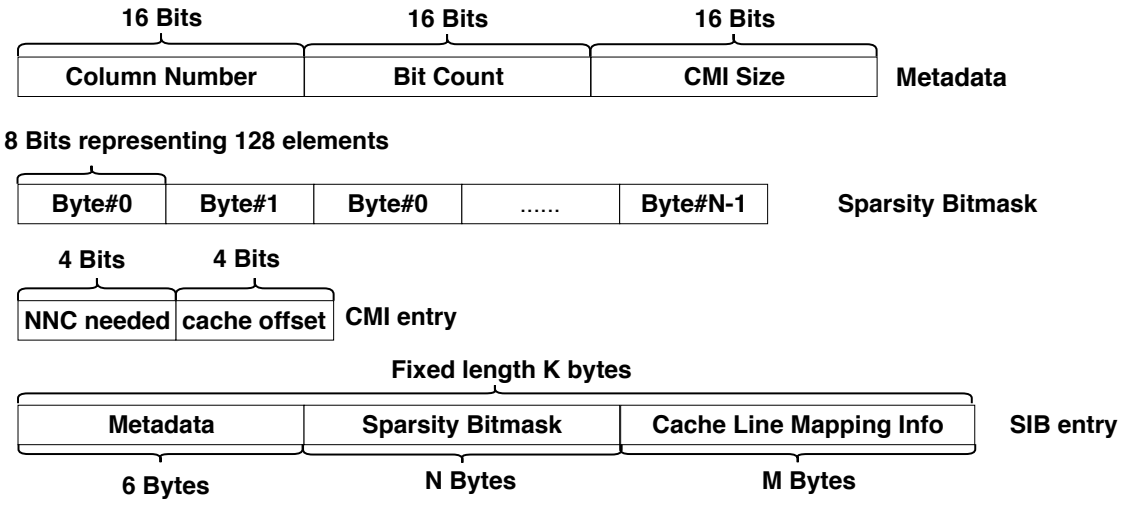


Figure 6.11: The overview of a SIB entry.

and M the CMI size in bytes. N can be determined by the column size col_size using the equation: $\lceil col\_size/128 \rceil$. K can be determined based on the column size as well. M can be calculated using $K - 6 - N$, which is also the maximum allowable number of CMI entries (a CMI entry only needs 1 byte). To determine K, we heuristically categorize the column size into one of four classes: 1) larger than 4096, 2) between 2048 and 4096, 3) between 1024 and 2048 and 4) smaller than 1024. We set K to 256, 128, 64 or 32 if the column size fits in category 1), 2), 3), or 4), respectively.

Given that CMI has a limited number of entries, we propose a sliding window replacement scheme. When the CMI is full, we remove the oldest CMI entry and keep the latest entries. By using this scheme, we keep the most recent records, taking advantage of the temporal locality.

6.4.3 Compaction Processor

The compaction processor contains multiple compaction processing units, servicing multiple prefetched data from main memory in parallel. Figure 6.12 shows the organization of the compaction processor. When activated, each compaction processing unit processes the data from an associated buffer (i.e., the data buffer shown in the figure) and then sends the compacted data to the prefetch buffer and the sparsity information to the SIB.

Figure 6.13 provides details on the compaction processing unit (ComPU). The ComPU is a special-purpose SIMD unit that performs two tasks: 1) FP32 to FP16 conversion, 2) comparisons with 0. The SIMD width is set to 16, as the memory requests to main memory are issued on a cache line basis. A fine-grained bitmask is generated according to the comparison results, reflecting the

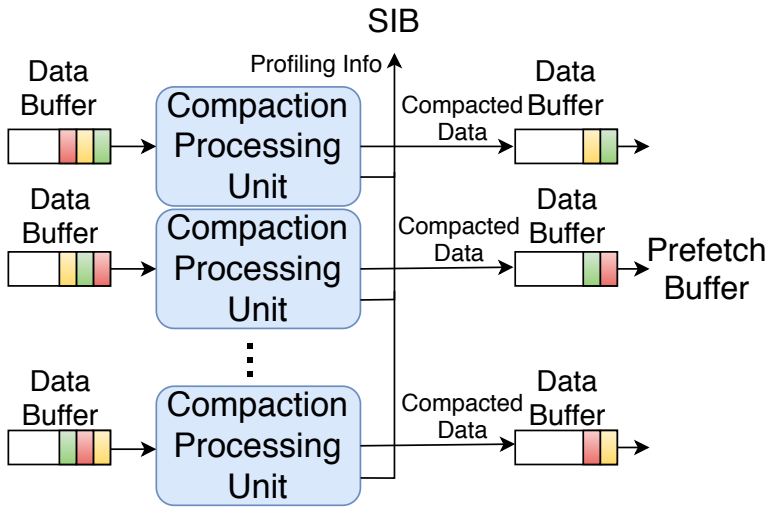


Figure 6.12: Overview of the compaction processor.

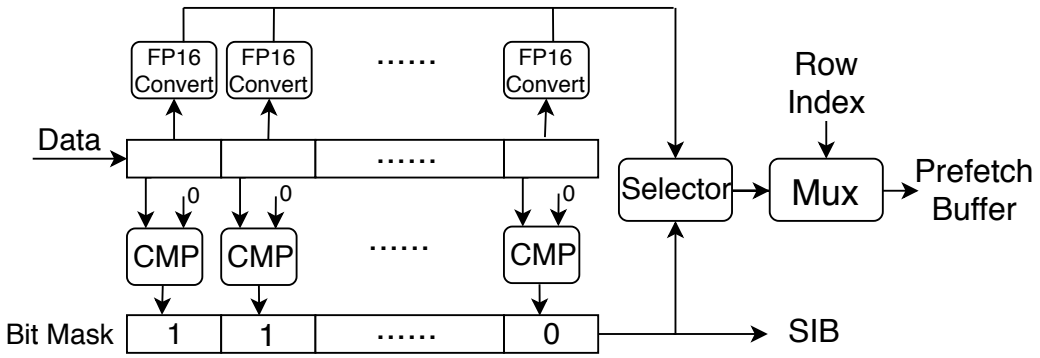


Figure 6.13: Overview of the compaction processing unit.

sparse pattern within the processed cache line. The ComPU uses this bitmask to select the non-zero elements, which are later bundled with the corresponding row index according to the ELLPACK-DIB format. In addition, the fine-grained bitmask is sent to the SIB as profiled sparsity information. Note that the row index can be calculated based on the memory transaction SIB issues.

6.4.4 Prefetch Buffer

The prefetch buffer has a multi-lane FIFO structure for storing data from the compaction processor. Each lane corresponds to an active SIB entry. During prefetching, the compacted data are stored in-order in the prefetch buffer. Once the prefetch buffer has collected one compacted cache line (i.e., 16 bundled elements) for a memory request, the data is sent to the L2. Due to prefetching,

Parameter	Size/Number
Maximum History Length	10
Sparsity Threshold	0.3
Initial Monitoring Period	500
Hibernation Monitoring Period Unit	10000

Table 6.2: Specifications of the sparsity monitor.

sometimes the compacted data is stored in the prefetch buffer before the sparse memory request arrives. We keep this data in the buffer. By doing so, we can take advantage of the earlier prefetch to further reduce data fetching latency.

6.5 Experimental Methodology

We evaluate the improvements due to the hardware-based compaction engine using a state-of-the-art GPU simulator. We also evaluate the overhead of the software-based sparsity monitor on real hardware.

Our experimental setup for evaluating the sparsity monitor uses two different classes of heterogeneous systems, targeting the training of DNN models at different scales. Both systems are equipped with both a CPU and a GPU. The first one is a desktop-grade system, equipped with an AMD Radeon RX Vega56 [178] as the GPU platform and an Intel(R) Core(TM) i7-8700 as the CPU platform. We select VGGNet-11 [17] and ResNet-10 [19] as the DNN models and use the CIFAR-10 dataset [179] for training. The second platform is a server-grade system, equipped with an NVIDIA Tesla V100 [131] as the GPU and an Intel(R) Xeon(R) E5-2630 as the CPU. We select AlexNet [16], VGGNet-16 [17] and ResNet-18 [19] as the DNN models, and use the ImageNet dataset [15] for training. We implement the sparsity monitor on TensorFlow 1.4 [76]. Table 6.2 shows the values for the parameters of the sparsity monitor that were described in Algorithms 1 and 2.

We use MGPUSim [180] to model the compaction engine present in our baseline AMD Radeon Instinct MI6 [181] GPU. Table 6.3 lists the details of the MI6. Table 6.4 lists the specification of the modeled compaction engine. We consume half of the space (1MB) of the baseline L2 cache for the storage of data related to the compaction engine, introducing no additional overhead for storage. In practice, the L2 space partition and the compaction engine are enabled only when launching a

sparse GEMM kernel. The command processor can issue corresponding commands to notify the hardware. When launching a dense GEMM kernel, the entire L2 space is used by the kernel. In our experiments with the compaction engine enabled, the portion of the L2 space devoted to the compacted data is static. However, the size of the partition could be set dynamically – a direction for future work.

Next, we present the methodology used to evaluate the compaction engine. Our evaluation is presented on a layer-by-layer basis. First, we select only the computation of the convolutional layers in forward propagation in our experiments, because the computations of convolutional layers in backward propagation are transposed convolutions [182], resulting in a very similar pattern as compared to forward propagation. To demonstrate the effectiveness of the compaction engine, we first select the convolution filter with the largest filter size ($3 \times 3 \times 512 \times 512$) and three different activation sparsity levels (65%, 70% and 85%, which are levels observed during the training of VGGNet-11 and ResNet-10 on CIFAR-10 dataset). Then, we conduct an extensive evaluation of all convolutional layers of AlexNet, VGGNet-16, and ResNet-18, using the sparsity levels observed during the training on the ImageNet dataset. In these experiments, we collect data during training across a number of epochs (one epoch indicates one round of training involving all images in the dataset). We pause sparsity profiling of each monitored activation map when we detect stability (5 epochs when training with ImageNet, 2 epochs with CIFAR-10).

In our experiments we use two different types of input data, but with the same sparsity levels. The first is synthesized data that has random locality (i.e., *Synthetic*), where the non-zero data is uniformly distributed. The second is obtained from training on real world datasets (i.e., *Real*). Table 6.5 provides the details of our experiments. We select three layers (Layer A, Layer B and Layer C) as representative layers for the models, all with the same filter size. Layers A, B, and C correspond to the second convolutional layer from the fourth residual block of ResNet-10, the fifth convolutional layer of VGGNet-11 and the seventh convolutional layer of VGGNet-11, respectively. For AlexNet and VGGNet-16, we use Convx-y, where the x and y values correspond to the convolutional layer and sparsity level, respectively. For ResNet-18, we use Convx_y-z to represent the layers, where the x, y and z values correspond to the residual block, the convolutional layer within the residual block, and sparsity level, respectively. Our baseline is a highly optimized dense matrix multiplication kernel we have selected from the AMD APP SDK [183].

Parameter	Size/Number
CU	36
Shader Array	9
L1 Vector Cache	16KB 4-way per CU
L2 Cache	2MB (base line) 1MB (with compaction engine)
DRAM	4GB

Table 6.3: Specifications of the modeled AMD MI6 Instinct GPU.

Parameter	Size/Number
SIB	512KB
Prefetch Buffer	512KB
Compaction Processing Unit	32

Table 6.4: Specifications of the compaction engine.

Model	Name	Sparsity	Batch Size
ResNet-10	Layer A	65%	128/256
VGGNet-11	Layer B	70%	128/256
VGGNet-11	Layer C	85%	128/256
AlexNet	Convx-y	45%-65%	128
VGGNet-16	Convx-y	35%-55%	64
ResNet-18	Convx_y-z	30%-50%	64

Table 6.5: Experimental setup for evaluating the compaction engine.

6.6 Results and Analysis

Next, we present the evaluation results for both the sparsity monitor and the compaction engine.

Table 6.6 provides the execution time overhead of using exhaustive profiling (i.e., measur-

Model Type	VGGNet-11	ResNet-10	AlexNet	VGGNet-16	ResNet-18
Dataset	CIFAR-10		ImageNet		
Exhaustive Profiling	14.9%	49.4%	21.4%	46.4%	94.2%
Profiling with the Sparsity Monitor	0.3%	0.6%	1.2%	5.7%	0.9%

Table 6.6: Overhead of two types of sparsity profiling: 1) exhaustive profiling and 2) sparsity monitor profiling. Overhead is reported relative to DNN training without any sparsity profiling.

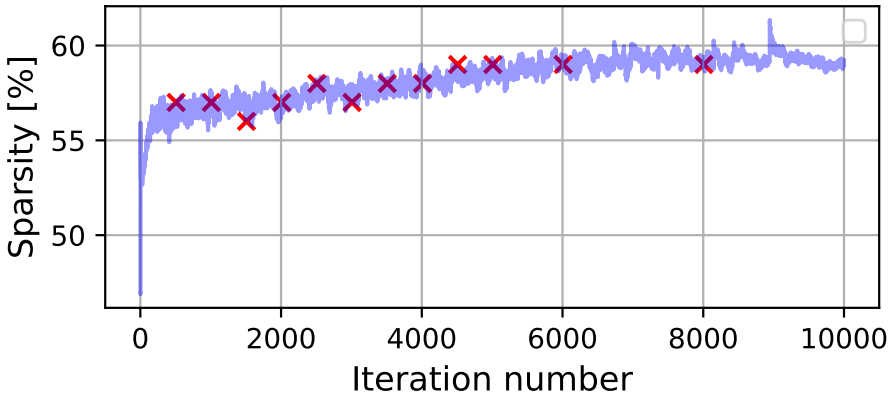


Figure 6.14: The profiling process of the sparsity monitor (VGGNet-11).

ing sparsity every training iteration) and profiling with our sparsity monitor. Our baseline is a DNN training process without performing any sparsity profiling. The training process runs for 50,000 training iterations for VGGNet-11 and ResNet-10, and 150,000 training iterations for AlexNet, VGGNet-16, and ResNet-18. From the table, we can see that profiling with the sparsity monitor has a limited overhead when training the DNN models (a maximum of 5.7%).

Figure 6.14 shows the data sparsity monitoring over time for the input of the fourth convolutional layer in VGGNet-11. Figure 6.15 shows the same profiling process at the input of the second convolutional layer in AlexNet. In these figures, the blue line shows the sparsity trend. The $\times$ symbols show when discrete sparsity measurements are collected by the sparsity monitor. From these results, we can see that the sparsity monitor is able to capture sparsity trends quite closely. The monitoring period (i.e., the distance between two $\times$'s) is extended whenever the sparsity monitor detects more stability in the degree of sparsity.

In terms of the performance improvements obtained using the compaction engine, we first

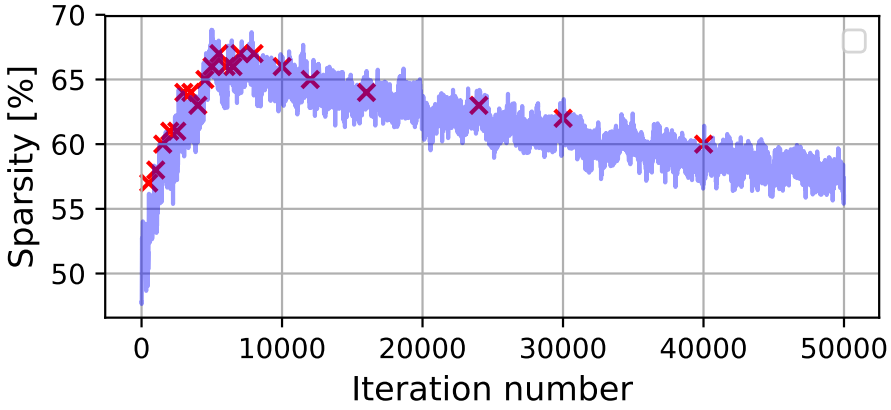


Figure 6.15: The profiling process of the sparsity monitor (AlexNet).

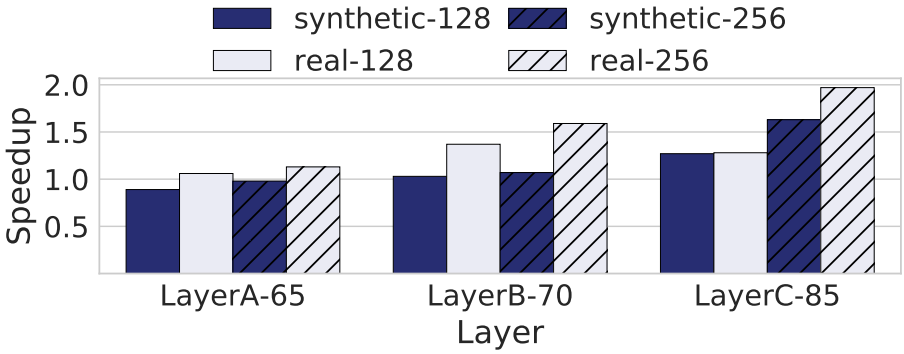


Figure 6.16: The performance improvements for layers A, B, and C with configurations described in table 6.5

present the speedup in Figure 6.16. Monitoring the activation maps from training VGGNet-11 and ResNet-10 with the CIFAR-10 dataset (real data), we can achieve a 1.24× average speedup using a batch size of 128, and achieve 1.56× average speedup with a batch size of 256.

Next, in Figures 6.17, 6.18 and 6.19 we show the performance improvements obtained for all the convolutional layers fed with sparse inputs for AlexNet, VGGNet-16 and ResNet-18, respectively. Inspecting these figures, when training with the activation maps using the ImageNet dataset (real data), we can achieve an average speedup of 3.4× for AlexNet, 2.14× for VGGNet-16, and 2.02× for ResNet-18.

From these results, we highlight two interesting observations: 1) Given the same problem size (same filter size, and number of input and output channels), the speedup increases when the

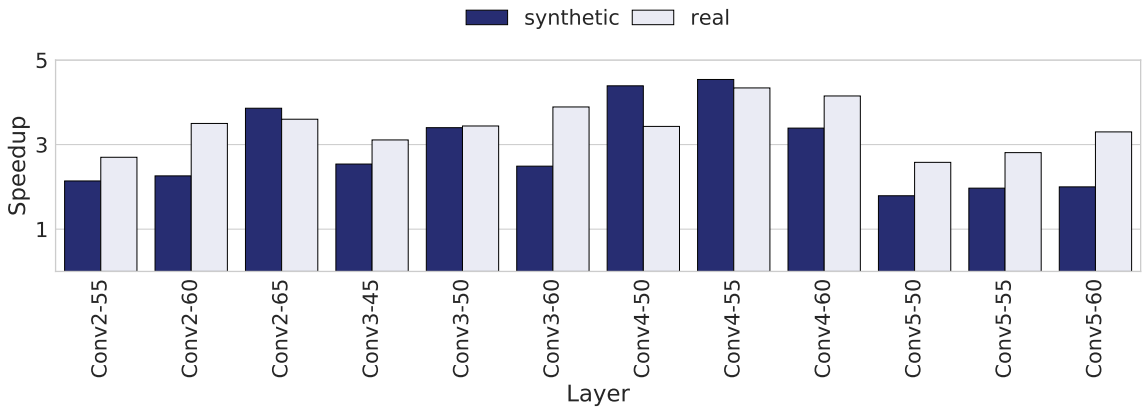


Figure 6.17: Performance improvements of convolutional layers in AlexNet.

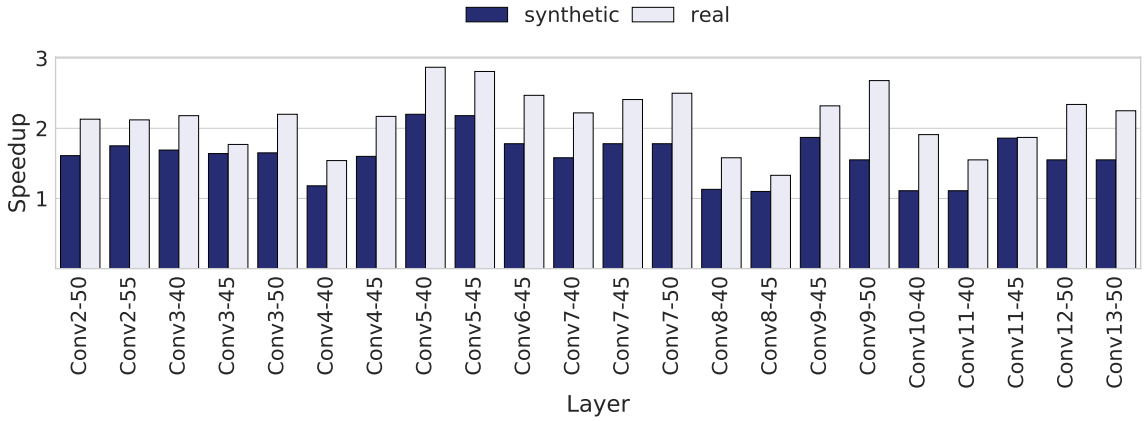


Figure 6.18: Performance improvements of the convolutional layers in VGGNet-16.

batch size grows, and 2) using data from the actual training, performance is superior to when using synthetic data. We will present a detailed analysis of these results in the next section.

6.6.1 Performance Analysis

To demonstrate the effectiveness of the compaction engine, we evaluate performance using three L2-related performance metrics: i) the number of memory transactions performed in L2, ii) the L2 hit rate, and iii) L2 read latency. We use normalized values for metrics i) and iii), relative to the value obtained when using dense GEMM (labeled as ”dense”). We use the absolute value for metric ii). We show results from the Layer A, B, and C in VGGNet-11 and ResNet-10, and all layers in AlexNet. For layer A, B, and C, we vary the batch size from 128 to 256. For the layers in AlexNet, we

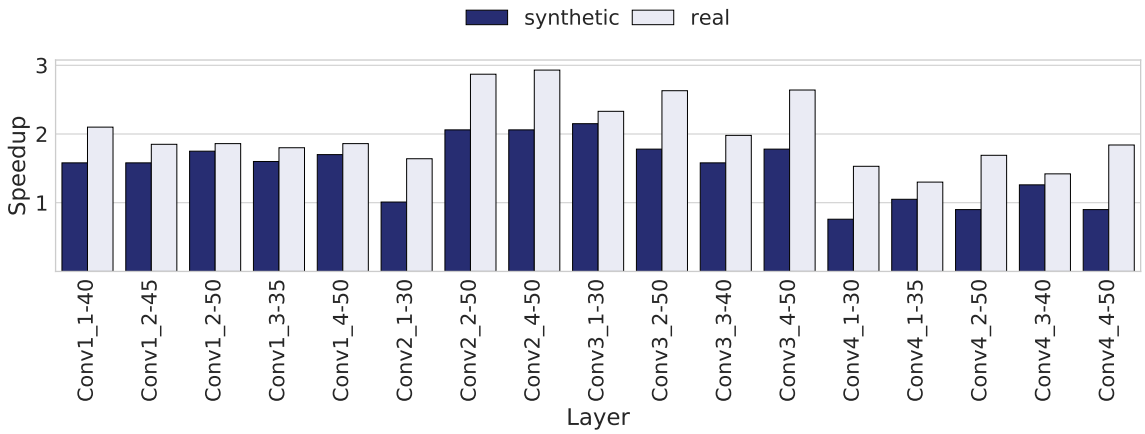
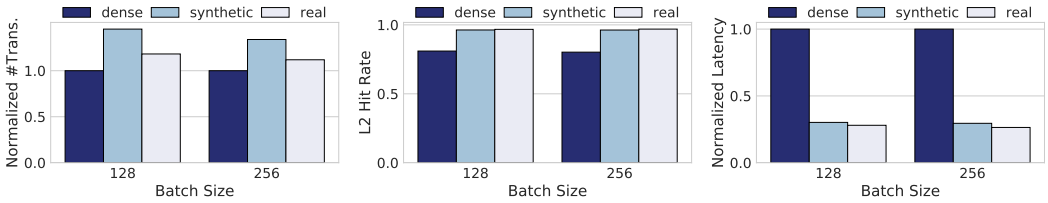
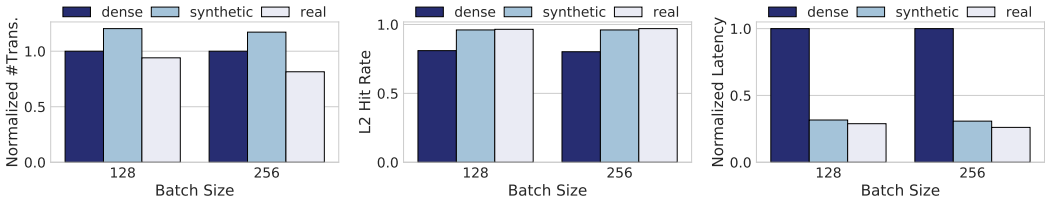


Figure 6.19: Performance improvements of the convolutional layers in ResNet-18.



(a) (b) (c)

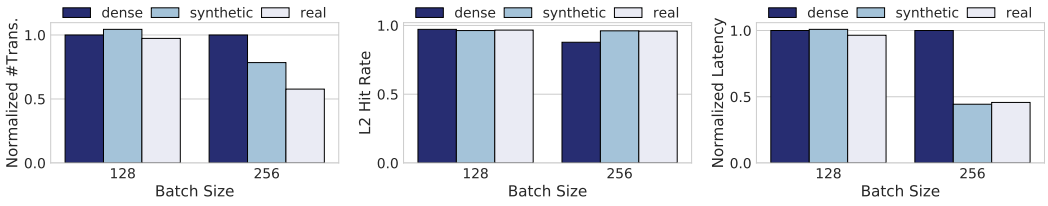
Figure 6.20: Performance metric of Layer A: (a) Normalized number of transactions.(b) L2 hit rate. (c) Normalized L2 read latency.



(a) (b) (c)

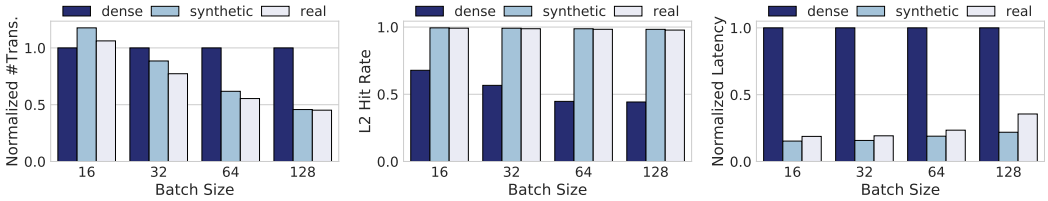
Figure 6.21: Performance metric of Layer B: (a) Normalized number of transactions.(b) L2 hit rate. (c) Normalized L2 read latency.

vary the batch size from 16 to 128. Inspecting Figures 6.20a-6.22c, we can see that the compaction engine reduces the number of memory transactions and reduces the memory access latency for L2, while increasing the L2 hit rate, especially when the batch size grows. The trend becomes more evident in Figures 6.23a-6.26c, where we use the ImageNet dataset. First, the compaction



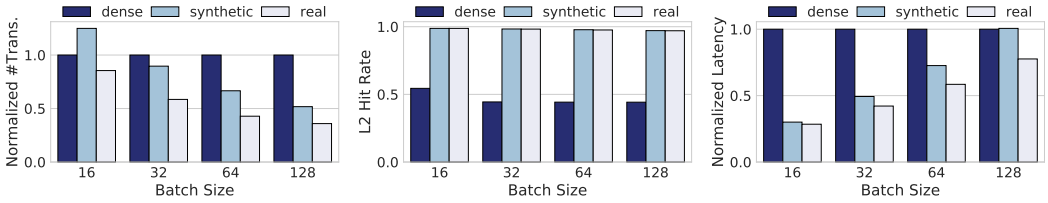
(a) (b) (c)

Figure 6.22: Performance metric of Layer C: (a) Normalized number of transactions.(b) L2 hit rate. (c) Normalized L2 read latency.



(a) (b) (c)

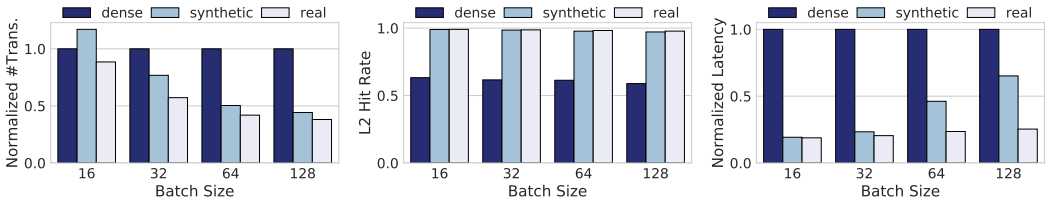
Figure 6.23: Performance metric of Conv2 in AlexNet: (a) Normalized number of transactions.(b) L2 hit rate. (c) Normalized L2 read latency.



(a) (b) (c)

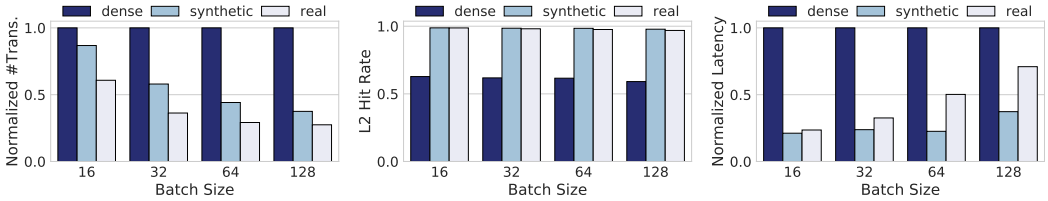
Figure 6.24: Performance metric of Conv3 in AlexNet: (a) Normalized number of transactions.(b) L2 hit rate. (c) Normalized L2 read latency.

engine enables us to leverage our proposed sparse GEMM, which results in fewer memory accesses. Even though the sparse GEMM algorithm leads to poorer data locality, this has little impact on the overall performance, especially when using real-world datasets for training. Second, the pattern of non-zero elements in the activation maps is regular and consecutive, as a result of the compaction engine, leading to a higher L2 hit rate with fewer memory accesses. Lastly, the compaction engine's prefetching mechanism significantly reduces the average access latency, enabling the compacted data



(a) (b) (c)

Figure 6.25: Performance metric of Conv4 in AlexNet: (a) Normalized number of transactions.(b) L2 hit rate. (c) Normalized L2 read latency.



(a) (b) (c)

Figure 6.26: Performance metric of Conv5 in AlexNet: (a) Normalized number of transactions.(b) L2 hit rate. (c) Normalized L2 read latency.

to be efficiently stored in a prefetch buffer. From our results we see that the compaction engine can achieve better memory performance when trained using real-world datasets. This can be explained by the fact that the real-world data captures the locality patterns present better than the synthetic data. In figures showing the normalized number of memory transactions, we noticed that the number drops when the batch size increases. We find that dense GEMM suffers more with increased batch size. When we increase the batch size to be n times larger, the activation maps grow by a factor of n. The compaction engine can effectively alleviate the extra pressure placed on the memory system given the same hardware configuration, reducing the number of memory accesses and caching the compacted data in the prefetch buffer.

6.6.2 DNN Training Speedup Modeling and Estimation

In this section, we discuss our model to accelerate DNN model training and estimate the benefits of Spartan for DNN training.

One challenge we experienced in this work is that the simulation of a full model training is extremely time consuming. Even equipped with a state-of-the-art GPU simulator such as MG-

PUSim [180], one training iteration of a full DNN model can take more than a day to complete. Simulating one epoch of training using the ImageNet dataset (10,000 iterations) would take more than 27 years to complete. But running simulation of a full model is not needed. As pointed out in previous DNN acceleration studies [58], the convolutional layers dominates the execution time. The speedup of the convolutional layers can alone serve as a reliable predictor of the overall training performance.

To estimate the overall training performance, we capture a model to calculate the speedup for DNN training using the compaction engine. We formalize the model in Equation 6.1. Equipped with this model, we can easily compute the potential benefits of Spartan across an arbitrary number of training iterations.

$$\frac{1}{O + \sum_{i}^{N} \sum_{j}^{M_i} P \frac{C_i R_{ij}}{S_{ij}}} \tag{6.1}$$

Similar to the strong scaling of Amdahl's law, we propose a speedup model that also consists of two terms, P and O. The P term represents the percentage of the total GPU kernel execution that is due to the convolutional layers in the DNN. The O term is the contributions due to other execution, including the time spent in other layers, overhead of the GPU kernel launch, and overhead of the sparsity monitor. C_i is the proportion of execution time for each convolutional layer, where $\sum_i^N C_i = 1$ and N is the number of convolutional layers in a DNN. R_{ij} is the detected sparsity level across the entire training, where $\sum_j^{M_i} R_{ij} = 1$ is for the ith convolutional layer and M_i is the detected sparsity level. S_{ij} is the speedup of the ith convolutional layer given j^{th} sparsity level.

We measure P and O values using DNNMark [68], while considering the overhead of the sparsity monitor. Table 6.7 shows the P values collected when training AlexNet, VGGNet-16, and ResNet-18 for the ImageNet dataset, across 5 epochs. The O values can be calculated by computing $1 - P$. We also measure the C_i using DNNMark and R_{ij} using the sparsity monitor. We use the speedup obtained from the simulator as S_{ij}. Table 6.7 also includes the estimated speedup (2.29×, 1.87×, and 1.55×) for the training for the AlexNet, VGGNet-16, and ResNet-18, respectively, using the ImageNet input dataset, across 5 epochs.

6.6.3 Hardware Costs

Given that Spartan is a software-hardware co-designed solution, we also need to evaluate the hardware cost involved. As a major component of Spartan, the sparsity monitor is implemented

	AlexNet	**VGGNet-16**	**ResNet-18**
P	87.40%	88.58%	82.20%
Speedup	2.29	1.87	1.55

Table 6.7: Measured P values and estimated speedup for training of AlexNet, VGGNet-16, and ResNet-18, with the ImageNet dataset, across 5 epochs.

in software, so no hardware cost is incurred. The only component that incurs hardware cost is the compaction engine. As the storage required by compaction engine (see Table 6.4) shares the space with the L2 cache, no additional hardware cost is needed for storage (though there is some minor performance overhead due to the logic for data look-up). For the processing part of the compaction engine, we need one 1-bit shifter and address calculator (one multiplier and an adder) for the SIB. We need 16 comparators, 16 FP16 converters, a 16-bit register, a selector and a multiplexer for each compaction processing unit.

Chapter 7

Exploring GPU Acceleration of DNNs using Block Circulant Matrices

Currently, the trend in DNN models (e.g., VGG [17], Googlenet[18], and Resnet[19]) is to keep adding additional parameters and layers, deepening, and widening the model to achieve better accuracy. As a result, the current computing platforms are facing growing demands on computing resources, memory and interconnect bandwidth, and storage resources.

Researchers have been pursuing new methods to improve the efficiency of DNN execution to overcome these challenges. Compressing the network model is one popular method. One way to compress the model is to use structured weight matrices, such as *circulant matrices* [184, 90]. FPGAs and ASICs have been the primary target in these studies [184, 90, 185, 186]. The main motivation for using structured matrices has been to improve energy efficiency, while also reducing execution time (especially in embedded applications for IoT products) during inference. However, training is also an integral part of deep learning. Structured matrices based weight compression techniques can benefit training as well, reducing the number of computations and the storage space for weights. FPGAs/ASICs are limited in their ability to be used effectively during DNN training.

GPUs, on the other hand, are a natural fit for accelerating the training of DNNs, especially when working with tremendous amounts of data (e.g., image classification with data from ImageNet [15]). GPUs have a large number of parallel arithmetic units, allowing thousands of threads to be launched concurrently, while also hiding memory access latency. For this reason, GPUs have been widely deployed to perform training of DNNs. In some settings, DNN training leverages multiple GPUs in a distributed manner [87]. However, naively applying a weight compression technique to

DNN training on a GPU can be highly inefficient, even causing a significant slowdown.

In this chapter, we explore DNN training acceleration on a GPU, using Block Circulant Matrices (BCMs) [132] for weight compression. BCMs can be used to replace the weight matrices in layers leveraging GEMM (e.g., convolutional and fully-connected layer). As a result, we replace GEMM computation with the BCM algorithm.

A BCM consists of many square circulant matrix blocks, as shown in Figure 7.1. The block size determines the compression ratio, which presents a tradeoff between the performance and model accuracy when approximating the desired matrices. The circulant matrices enable the transformation of regular matrix multiplications to *circulant convolutions*, reducing computing complexity by employing Fast Fourier Transforms (FFTs) and Inverse Fast Fourier Transforms (IFFTs).

To study the BCM algorithm for DNN training on a GPU, we first analyze the algorithmic steps in the BCM algorithm. We characterize three different scenarios, considering forward and backward propagation. We analyze the algorithmic complexity of each stage for the different scenarios, highlighting some of the challenges when leveraging BCMs on a GPU. Next, perform both general and GPU-specific optimizations that impact: i) the decomposition and interaction of individual operations, and ii) the overall GPU kernel design. We modify the algorithmic steps to remove redundant computations, while maintaining mathematical integrity. We also leverage multiple GPU kernel optimizations, considering performance factors, such as occupancy, data sharing/reuse patterns, and memory coalescing.

Our proposed methods can benefit all types of DNNs leveraging GEMM. The computation involved in the fully-connected layer is a GEMM operation. The convolutional layer can use GEMM with an additional im2col operation [90]. The computations in an LSTM unit [187] are also GEMM operations.

7.1 Block Circulant DNN

It has been formally proven that some structured matrices have the universal approximation property [188, 189]. Among them, circulant matrices can be used in constructing neural networks given their properties to reduce both the amount of storage and computing complexity through FFTs/IFFTs [132]. Block circulant weight matrices have been proposed [90, 184, 190] to address the two limitations of a circulant matrix: 1) it must be square, and 2) the compression ratio is invariant, resulting in a high degree of information loss for large matrices.

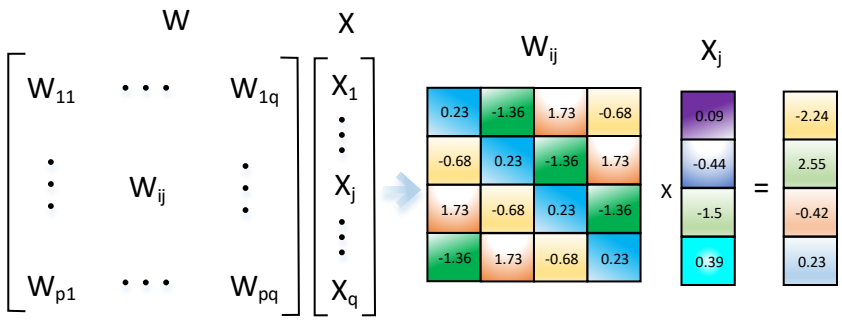


Figure 7.1: An example of a Block Circulant Matrix (BCM).

A BCM is composed of an array of equally-sized square circulant matrices (i.e., blocks), whose size is configurable. This method carefully considers the tradeoff between compression ratio and model performance by selecting the best block size (i.e., a larger block size corresponds to a larger compression ratio and improved computation speed, though a smaller block size corresponds to a higher accuracy).

The convention to describe a BCM is as follows: $\mathbf{W}$ represents the block circulant weight matrix, $\mathbf{X}$ the input, and $\mathbf{a}$ the output. Within $\mathbf{W}$, p denotes the number of blocks row-wise, q the number of blocks column-wise, and k the block size, which is the number of free elements in one circulant matrix block (i.e., $\mathbf{W}_{ij}$, with a total of k^2 elements). Figure 7.1 shows an example of $\mathbf{W} * \mathbf{X}$, a circulant matrix block $\mathbf{W}_{ij}$, and one corresponding input vector $\mathbf{X}_j$, in which $i = \{1...p\}$ and $j = \{1...q\}$. Using a BCM, the number of weights needed for each linear transformation layer reduces from $p * q * k^2$ to $p * q * k$.

7.1.1 Training with BCM Algorithm

The DNN training process involves forward and backward propagations. The computation of the forward propagation is $\mathbf{W} * \mathbf{X}$. As the weight matrix is now block circulant, the computation involves two steps that reduce the overall computational complexity. First, within each block, GEMMs are transformed into an "FFT $\rightarrow$
element-wise multiplication $\rightarrow$ IFFT", according to the *circulant convolution theorem* [191, 192]. Second, the results over index j can be summed to produce the final result.

Backward propagation involves two operations both using $\frac{\partial \mathbf{L}}{\partial \mathbf{a}}$. The first computes the derivative for the inputs $\frac{\partial \mathbf{L}}{\partial \mathbf{X}}$ and the second operation computes the derivative for the weights $\frac{\partial \mathbf{L}}{\partial \mathbf{W}}$.

In the context of the BCM algorithm, $\frac{\partial \mathbf{a}}{\partial \mathbf{W}}$ and $\frac{\partial \mathbf{a}}{\partial \mathbf{X}}$ are both BCMs [132]. Within the scope of each block, $\frac{\partial \mathbf{a}_i}{\partial \mathbf{W}_{ij}}$ is a circulant matrix defined by vector $\mathbf{X'}_j : \{X_{j1}, X_{jk}, X_{j,k-1}, ..., X_{j2}\}$ and

$\frac{\partial \mathbf{a}_i}{\partial \mathbf{X}_j}$ is a circulant matrix defined by $\mathbf{w'}_{ij} : \{w_{ij1}, w_{ijk}, w_{ij,k-1}, ..., w_{ij2}\}$.

Usually, every training iteration should consume one batch of input data, producing one batch of outputs. In this chapter, we consider training in batches and develop a set of equations to describe batched DNN training. To characterize the underlying data format, we use index b to indicate the position of data within each batch and n to denote the number of input data elements in one batch. Equations for the forward and backward propagations are shown below:

$$\mathbf{a}_{bi} = \sum_{j=1}^{q} IFFT(FFT(\mathbf{w}_{ij}) \circ FFT(\mathbf{X}_{bj})) \tag{7.1}$$

$$\frac{\partial \mathbf{L}}{\partial \mathbf{W}_{ij}} = \sum_{b=1}^{n} IFFT(FFT(\frac{\partial \mathbf{L}}{\partial \mathbf{a}_{bi}}) \circ FFT(\mathbf{X'}_{bj})) \tag{7.2}$$

$$\frac{\partial \mathbf{L}}{\partial \mathbf{X}_{bj}} = \sum_{i=1}^{p} IFFT(FFT(\frac{\partial \mathbf{L}}{\partial \mathbf{a}_{bi}}) \circ FFT(\mathbf{w'}_{ij})) \tag{7.3}$$

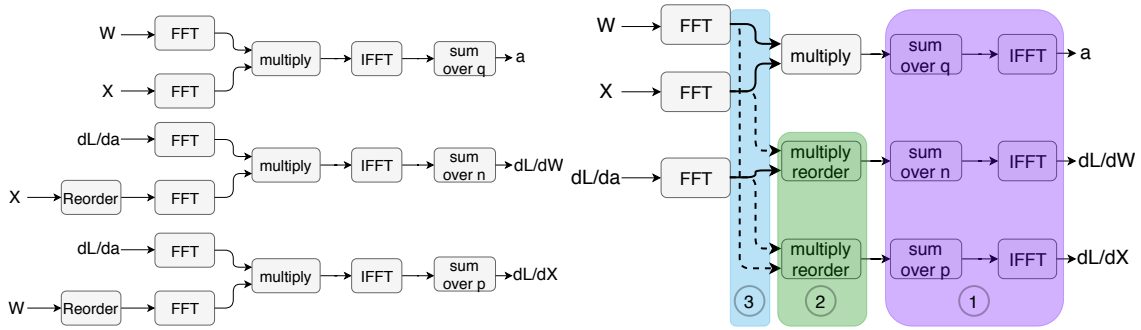
For simplicity, we set the row height and column width of the weight matrices to power of 2. If dimensions of the linear transformation layers are not power of 2, zero padding can be used both on the feature maps and weight matrices. Note that zero padding does not effect computation complexity and is used for regularizing the formats only.

7.2 Implementing BCM based DNN Training on a GPU

Next, we take a deep dive into the operations performed when training a linear transformation layer using BCMs. First, we consider how to map the BCM algorithm when targeting a GPU. Then, we consider some of the performance challenges present in our baseline GPU implementation.

7.2.1 Decomposing Forward and Backward Propagation

Considering Equations (7.1), (7.2), and (7.3), the computations associated with forward and backward propagation can be further broken down into multiple stages. Figure 7.2a shows the stages required during forward and backward propagations, where each block represents the operations in the equations and the arrows represent data dependencies between two blocks. For instance, the *multiply* block depends on the results computed by the two FFTs blocks. Whenever there is a dependency, synchronization between the two operations is required.



(a) (b)

Figure 7.2: (a) The flow of operations performed during a complete training. (b) An optimized implementation of the same operations.

7.2.1.1 Managing Data

All the data, including the intermediate results from each stage, are stored as tensors (multi-dimensional arrays) in GPU memory. The data layout in our baseline implementation uses a conventional tensor data layout. A tensor has N dimensions, denoted as D_i, where i can take on values $0, ..., N-1$. Each dimension has an associated size value M_i, specifying the number of data chunks for the specific dimension. Based on the memory layout, we can obtain the index used to access an element of the data by leveraging the dimension-associated indices and M_i. Taking the tensor data as an example, which has dimensions of $n \times p \times q \times k$, we can calculate the index to access any element using:

$$\begin{aligned} idx = & D3\_idx * p * q * k + D2\_idx * q * k \\ & + D1\_idx * k + D0\_idx \end{aligned} \tag{7.4}$$

, in which the $D3\_idx$ (0 to $n-1$), $D2\_idx$ (0 to $p-1$), $D1\_idx$ (0 to $q-1$), and $D0\_idx$ (0 to $k-1$) are indices associated with dimension D_3, D_2, D_1 and D_0, respectively.

The format of each input and output involved in the BCM algorithm is indicated in Table 7.1. The dimensions n, p, q and k are described in Section 7.1.

7.2.2 Kernels of the Operations

Inspecting the algorithm decomposition in the forward and backward propagation, we see four primary operations: 1) FFTs, 2) IFFTs, 3) element-wise multiplications, and 4) summations. The reorder kernel is already efficient, so we exclude it from this discussion.

Table 7.1: Dimensions of the input and output data.

Type	Name	Format
Inputs, Outputs	$W, \frac{\partial L}{\partial W}$	$p \times q \times k$
	$X, \frac{\partial L}{\partial X}$	$n \times q \times k$
	$a, \frac{\partial L}{\partial a}$	$n \times p \times k$

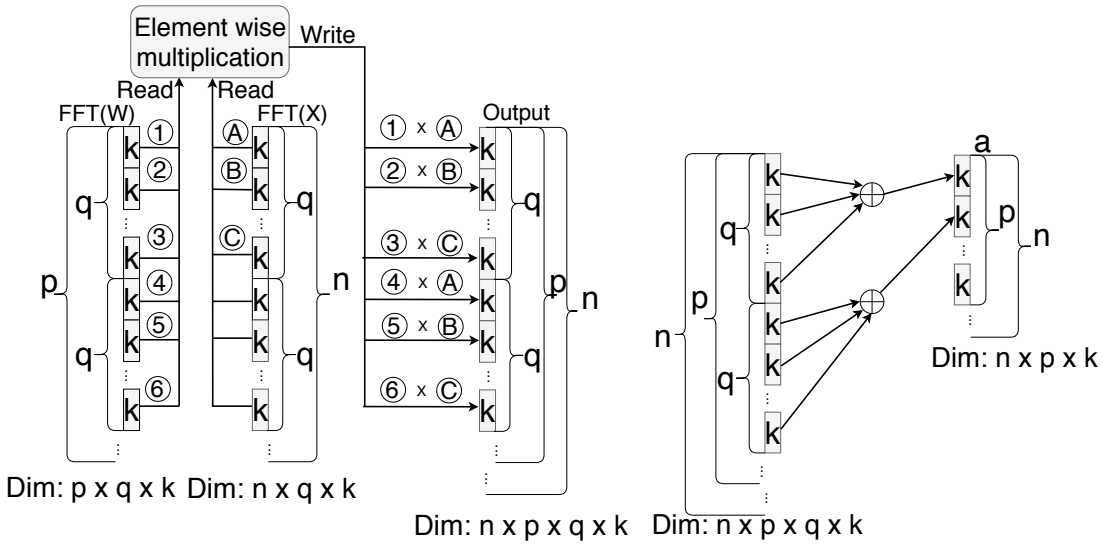


Figure 7.3: (a) Operational overview of element-wise multiplications performed during forward propagation. (b) Reduction summations performed during forward propagation.

To compute the FFT/IFFT efficiently, we select the cuFFT [193] library, a highly optimized FFT/IFFT library specifically designed for NVIDIA GPUs. In our implementation, we use the 1D FFT/IFFT, launched in batches over the inner-most dimension D_0.

Figures 7.3a and 7.3b illustrate the operations involved in the element-wise multiplications and summations during forward propagation, together with the indexing pattern associated with the memory layout. We omit the two operations performed during backward propagation as they share a similar pattern. The major differences are: 1) the format of the input, and 2) the locations read from the inputs.

Figure 7.3a illustrates the operational details for the element-wise multiplications during forward propagation. From the figure, we can see that element-wise multiplications are performed

over k-sized data chunks. The chunks are read from $FFT(W)$ and $FFT(X)$, with dimensions of $p \times q \times k$ and $n \times q \times k$, respectively. In the example presented in the figure, the labels ①-⑥ correspond to 6 different k-sized data chunks read from $FFT(W)$, and the labels Ⓐ-Ⓒ stand for the 3 different k-sized data chunks read from $FFT(X)$. The labels on the write path (e.g., ① × Ⓐ) identify the results from the element-wise multiplications (for data chunks ① and Ⓐ in this example). Likewise, the other labels on the write path are the result of element-wise multiplications for their respective data chunks.

Figure 7.3b shows the reductions (i.e., summations) performed during forward propagation. Given the input, which is the output shown in Figure 7.3a, the summation operations are performed over different dimensions, q for the forward propagation, n for the backward propagation (for the weights), and p for the backward propagation (for the data).

For the element-wise multiplications and summations, we design and implement GPU kernel functions. In contrast to an FPGA, a GPU has a very different hardware architecture. Thus, we arrive at a very different algorithmic implementation for these operations. The basic elements required for a GPU kernel include: i) mapping the data layout to the kernel dimensions (i.e., the shape of the cooperative thread array, defined by blockDim and gridDim [66]), ii) specifying the operations to be performed in the kernel, and iii) managing the kernel input data, stored in GPU memory while the kernel is active.

According to the data layout in the BCM algorithm, the data unit involved in the computations is a k-sized data chunk (i.e., the size of the inner-most dimension). Therefore, the most straightforward scheme would be to directly map both element-wise multiplications and summations to a space where the kernel parameters (i.e., blockDim.x, gridDim.x, gridDim.y, and gridDim.z) are determined by the tensor data dimensions. With this mapping mechanism, each thread block has k threads simultaneously running on a GPU. Given this mapping philosophy, the kernel becomes significantly simplified. Table 7.2 shows the mapping details.

7.2.3 Challenges

First, our decomposition of the algorithm shows that there are at least five stages for both forward and backward propagation. When executing this multi-stage application on a GPU, we need to launch multiple kernels sequentially due to the constraints of the dependencies between stages, adding the overhead of kernel launch and associated synchronizations to the overall execution time. As such, the benefits of parallelism on a GPU are limited to each kernel. BCM operations cannot

Table 7.2: Kernel dimensions for element-wise multiplications and summations (Forward Propagation).

Kernels	Dimension Variables	Forward
multiplication	blockDim	$(k, 1, 1)$
	gridDim	(q, p, n)
summation	blockDim	$(k, 1, 1)$
	gridDim	$(p, n, 1)$

benefit from concurrent kernel execution given the large size of the kernels.

The kernels executed after FFT and before IFFT, in the sequence of BCM operations, suffer from an occupancy issue. Given that we are using the NVIDIA cuFFT library kernels for FFT/IFFT execution, the size of k (the thread block size after kernel dimension mapping) changes to $k/2 + 1$ after the FFT, which is due to a memory-saving feature of the cuFFT library [193]. As a result, the thread block size is not always a multiple of 32 (warp size). This new value for k, denoted as k' in this chapter, results in lower occupancy and underutilization of the device.

Last but not least, a regular mapping function cannot generally exploit the capabilities of a GPU due to inefficient use of hardware components, such as the memory coalescing units and shared memory. The former coalesces multiple memory transactions issued from the same thread block, and the latter enables fast access to data that is frequently reused by multiple threads in a thread block [66]. The nature of the two customized kernels results in a complicated memory access pattern (i.e., tensor data indexing). Improving memory layout can significantly impact coalescing efficiency and improve performance. Our implementation of these two kernels also produces some predictable (and cacheable) data reuse patterns. For example, in Figure 7.3a, Ⓐ, Ⓑ, and Ⓒ are reused when writing the results over to dimension p.

7.3 Optimizations

7.3.1 Improving the Flow of Operations

In this section, we describe the set of optimizations we develop/apply to reduce the number of redundant operations in the BCM algorithm. Our optimizations preserve the mathematical integrity of all steps. As shown in Figure 7.2b, our optimizations consist of 3 enhancements identified in purple, green and blue, and numbered 1, 2, and 3, respectively. These enhancements include: 1)

swapping the position of the IFFT and summation operations, 2) removing the *reorder* operation and modifying the associated *multiplication* to guarantee mathematical correctness, and 3) reusing results after computing the FFT for backward propagations for weights and data. In the following paragraphs, we provide details of these steps and show how their use can simplify computations.

Step 1 does not impact the mathematical integrity of the computation, given that an IFFT is a linear transformation. The IFFT can be performed on A and B as individual components or combined in a sum, given that $IFFT(A + B) = IFFT(A) + IFFT(B)$. Therefore, we can swap the position of the *sum* and *IFFT* in the BCM algorithm. The benefit of these simple changes results in a significant reduction in terms of the number of computations needed for computing the IFFT. Similarly, the same optimizations can be applied to the backward propagation as well.

For step 2, the $\mathbf{w}'$ and $\mathbf{X}'$ are in fact the same vectors for $\mathbf{w}$ and $\mathbf{X}$, but shifted by $k - 1$ elements. Based on the fundamental characteristics of an FFT [194], the magnitude of $FFT(\mathbf{w}')$ and the magnitude of $FFT(\mathbf{w})$ are the same, with only the sign of the imaginary part flipped. Based on this property, we can remove the *reorder* operation entirely and only modify the associated element-wise multiplications, considering the flipped sign in the imaginary part of the $FFT(\mathbf{w})$ and $FFT(\mathbf{X})$. Applying step 2 leads directly to step 3. Removing the *reorder* operation adds some redundancy that we can exploit for further optimizations, specifically during backward propagation. In the blue shaded area ③ of Figure 7.2b, the data reuse paths are indicated using dashed arrows.

For the computation of $FFT(\mathbf{X})$ and the $FFT(\mathbf{w})$ during backward propagation, we can reuse the corresponding results produced during forward propagation. In addition, the results from the $FFT(\frac{\partial \mathbf{L}}{\partial \mathbf{a}})$ computation can also be reused during backward propagation. In all, we can eliminate two *reorder* operations and three major FFT operations during backward propagations, significantly reducing the number of operations.

7.3.2 Kernel Customizations

In this section, we present a number of kernel-specific optimizations for the two customized kernels, based on their execution performance characteristics. To guide these optimizations, we consider three different performance-related factors: 1) occupancy, 2) data reuse pattern, and 2) memory coalescing for both reads and writes. We develop a set of principles that are focused on these three factors. We provide a brief description of each of them.

- Principle I: To achieve high occupancy, the thread block size should be a multiple of 32 (the warp size).

- Principle II: Each thread block should include threads that exhibit temporal locality in their data access pattern.
- Principle III: Given a thread block design based on principles I and II, the memory access distance (defined later) within one thread block exploits memory coalescing.

Given the lack of control over thread scheduling on the actual hardware, ***mapping functions tuning*** (Section 7.2.2) and ***memory layout reorganization*** are the keys to applying our principles. To quantitatively evaluate the benefits of principles II and III, we define two metrics: i) the degree of data reuse (DR), and ii) the memory access distance (MD) for both reads and writes. Both metrics are measured within a single thread block. We define the DR metric as ***the maximum number of threads that reuse the same data***. A larger DR is preferred to enable better utilization of shared memory. We define the MD metric as ***the range of addresses accessed across all memory accesses, divided by the number of threads per thread block***. The MD metric is used to assess spatial locality. An ideal MD value would equal to 1 storage unit (a float or double), meaning that the memory accesses within one thread block are entirely regular and consecutive, leading to an ideal spatial locality. Any MD value larger than 1 storage unit implies some degree of irregular access. A more significant value for MD translates to a lower spatial locality. In this chapter, we focus on reducing the MD value by reorganizing the memory layout. Both metrics can be easily obtained through the given mapping function and memory layout of the input and output data. Note that tuning the DR and MD metrics is a tradeoff, in that increasing DR can increase MD while introducing irregular memory accesses.

7.3.2.1 Element-wise Multiplications

The element-wise multiplication kernel suffers from occupancy issues, such that data reuse and memory coalescing are heavily underexploited. To address these issues, we propose two optimizations (labeled O1 and O2), which both consider principles I and II, but in different priority orders. We also transform the memory layout to decrease MD, as is the goal with principle III.

The O1 optimization is aimed at addressing the occupancy issue first, then additionally leveraging the data reuse pattern to support better shared memory usage. We achieve this by directly increasing the size of a thread block, including more threads from a second dimension rather than the innermost dimension. More specifically, with the mapping functions indicated in Table 7.2, given a multiple of 32 for N, we can change the mapping of gridDim.x to $(q - N + 1)/N$, while

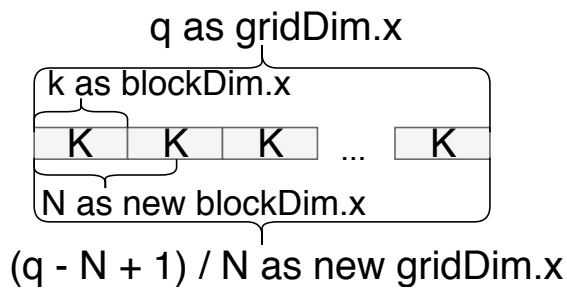


Figure 7.4: Kernel dimension mapping with a new thread block size.

keeping the gridDim.y and gridDim.z dimensions unchanged. Figure 7.4 illustrates this change in the kernel dimension mapping. However, this can produce an undesirable MD value during backward propagation, resulting in long-strided memory accesses because the two inputs for the multiplication do not share the same innermost dimensions. To address this issue, we reorganize the memory layout, leading to a better chance of memory coalescing. For example, during backward propagation for weights, the dimensions for the two inputs for the multiplication are $n \times p \times k$ and $n \times q \times k$. We reorganize them so that their dimensions become $p \times n \times k$ and $q \times n \times k$.

Likewise, the memory layout of their outputs need to be reorganized as well, in order to avoid large MD values for the writes.

To leverage any temporal reuse pattern in the data, we manipulate the mapping function as well. We keep the blockDim.x unchanged as N, and map DR to the blockDim.y dimension, increasing the number of threads that share data. Given the constraint that a single thread block can only contain 1024 threads [66], DR can be calculated using Equation 7.5. This optimization approach considers a tradeoff. A larger DR value can make better use of the shared memory but results in a smaller blockDim.x dimension, potentially leading to lower spatial locality within one thread block. From our experimental results, we found that the blockDim.x can be selected using the following rule of thumb: if k' is less than 32, then we set the blockDim.x to 32. Otherwise, we set blockDim.x to k.

Algorithm 6 describes the steps required to map kernel O1, and includes reorganization of the memory layout and initialization of kernel parameters.

$$DR = \frac{1024}{blockDim.x} \tag{7.5}$$

$$DR = \frac{1024}{k} \tag{7.6}$$

Algorithm 6 Kernel Mapping Function of O1

1: **Inputs:** n, p, q, k, $FFT(\frac{\partial L}{\partial a})$, $FFT(W)$, $FFT(X)$, propagation_type
2: **Outputs:** blockDim, gridDim
3: **if** propagation_type == BACKWARD_WEIGHT **then**
4: MemoryReorganize($FFT(\frac{\partial L}{\partial a})$) $\rightarrow p \times n \times k$
5: MemoryReorganize($FFT(X)$) $\rightarrow q \times n \times k$
6: M_1, M_2, M_3 = n, q, p
7: **else if** propagation_type == BACKWARD_DATA **then**
8: MemoryReorganize($FFT(W)$) $\rightarrow q \times p \times k$
9: M_1, M_2, M_3 = p, q, n
10: **else if** propagation_type == FORWARD **then**
11: M_1, M_2, M_3 = q, p, n
12: **end if**
13: k $<$ 32 ? N = 32 : N = k
14: blockDim.x = N
15: gridDim.x = $(M\_1 - N + 1)/N$
16: DR = $\frac{1024}{blockDim.x}$
17: blockDim.y = DR
18: gridDim.y = $\frac{M\_2 - DR + 1}{DR}$
19: gridDim.z = M_3

The O2 optimization, on the other hand, focuses on leveraging the data reuse pattern first. We calculate DR using Equation 7.6. Note that the blockDim.x is k' in O2, computed as $k/2 + 1$. Equation 7.6 is, in fact, derived from a different process. Given that k' is not a multiple of 32, we need to find a DR value using two constraints: 1) the DR value has to be a power of 2, and 2) $DR * (k/2 + 1) <= 1024$. The DR value should be the largest power of 2 which satisfies $DR <= \frac{2048}{k+2} < \frac{2048}{k}$, leading to Equation 7.6.

After adopting this approach, we find that there is no need to increase the thread block size to improve occupancy, as the thread block size equals to $\frac{1024}{k} * (k/2 + 1)$, which can be further expanded as $512 + \frac{1024}{k}$. Assuming that we only select a power of 2 for k, the new thread block size is a multiple of 32 when $k <= 32$. Even when k's value is greater than 32, the impact is negligible, thanks to the constant term (512) for the thread block size.

Algorithm 7 Kernel Mapping Function of O2

1: **Inputs:** n, p, q, k, $FFT(\frac{\partial L}{\partial a})$, $FFT(W)$, $FFT(X)$, propagation_type
2: **Outputs:** blockDim, gridDim
3: **if** propagation_type == BACKWARD_WEIGHT **then**
4: M_1, M_2, M_3 = n, q, p
5: **else if** propagation_type == BACKWARD_DATA **then**
6: M_1, M_2, M_3 = p, q, n
7: **else if** propagation_type == FORWARD **then**
8: MemoryReorganize($FFT(W)$) $\rightarrow p \times k \times q$
9: M_1, M_2, M_3 = q, p, n
10: **end if**
11: k' = k / 2 + 1
12: blockDim.x = k'
13: gridDim.x = M_1
14: DR = $\frac{1024}{k}$
15: blockDim.y = DR
16: gridDim.y = $\frac{M\_2 - DR + 1}{DR}$
17: gridDim.z = M_3

The backward propagations have smaller MD values when applying O2, versus forward propagation. The MD value for the former is 1, which is ideal, whereas the latter is $q * k'/N$ approximately. When $N < q * k'$ (a common scenario), so the MD is larger than 1. As such, we reorganize the memory layout of W from $p \times q \times k$ to $p \times k \times q$ for forward propagation.

Algorithm 7 describes the steps required to map kernel O2, including reorganization of the memory layout and initialization of kernel parameters.

Comparing with O1, O2 has both advantages and disadvantages. In terms of advantages, first, O2 only needs one memory reorganization operation, while O1 needs three. Second, O2 has a smaller MD value for reads because of the same reason described in the previous paragraph. In terms of disadvantages, O2 has a larger MD value for writes (we need to reorganize the output for the reduction summation). The occupancy issues for O1 and O2 are no longer existing. Achieving a better DR value is a bit tricky for these two optimizations, as the value for DR depends on the size of k. When $k < 32$, O2 achieves a better DR value, whereas, O1 and O2 have similar data reuse

Algorithm 8 Kernel Mapping Function of Reduce Summation

1: **Inputs:** n, p, q, k, propagation_type
2: **Outputs:** blockDim, gridDim
3: **if** propagation_type == BACKWARD_WEIGHT **then**
4: M_1, M_2 = q, p
5: **else if** propagation_type == BACKWARD_DATA **then**
6: M_1, M_2 = q, n
7: **else if** propagation_type == FORWARD **then**
8: M_1, M_2 = p, n
9: **end if**
10: k' = k / 2 + 1
11: blockDim.x = k'
12: gridDim.x = M_1
13: gridDim.y = M_2

behaviors when $k >= 32$.

7.3.2.2 Reduce Summation

Based on the execution patterns and memory access patterns illustrated in Figure 7.3b, the kernels only perform a simple summation of the data along a single dimension, resulting in no data reuse. Therefore, we only focus on optimizing the occupancy and MD value for the summation. The baseline mapping function results in undesirable MD values for backward propagations. To improve spatial locality, we reorganize the memory layout of the outputs of the previous kernels (i.e., the element-wise multiplications) so that the specific dimension for performing the summation becomes the second innermost dimension. To address the occupancy issue, we can also use the same approach illustrated in Figure 7.4 to increase the thread block size. However, our experimental results show that this can lead to irregular memory access patterns, impacting spatial locality. Therefore, we only apply memory layout reorganization to improve MD on reads for the summations.

Algorithm 8 describes the steps required to map the Reduce Summation kernel. The reduce summation does not require changes to the memory layout since the element-wise multiplication has already remapped the memory layout to a desired format.

Algorithm 9 Kernel Mapping Function of Kernel Fusion Method

1: **Inputs:** n, p, q, k, propagation_type
2: **Outputs:** blockDim, gridDim
3: **if** propagation_type == BACKWARD_WEIGHT **then**
4: M_1, M_2 = q, p
5: **else if** propagation_type == BACKWARD_DATA **then**
6: M_1, M_2 = q, n
7: **else if** propagation_type == FORWARD **then**
8: M_1, M_2 = p, n
9: **end if**
10: k' = k / 2 + 1
11: blockDim.x = k'
12: gridDim.x = 1
13: DR = $\frac{1024}{k}$
14: blockDim.y = DR
15: gridDim.y = $\frac{M\_1 - DR + 1}{DR}$
16: gridDim.z = M_2

7.3.2.3 Kernel Fusion

Returning to our optimized implementation for the BCM algorithm, the first step swaps the position of the IFFT and the summation to reduce the amount of computation required during the IFFT. The optimized BCM algorithm provides us with an opportunity to ***fuse*** these two kernels into one. We apply kernel fusion, considering the same design principles just described, and leverage the same optimizations. We implement a new mapping function for the fused kernels based on O2. We chose O2 over O1 because O2 does a better job of improving the locality of the reads. We omit the memory layout reorganization operations as they are no longer necessary because the performance gains from reorganizing memory layout are negligible after the kernel fusion.

Algorithm 9 describes the steps required to map the Kernel Fusion method.

The kernel fusion approach has the following potential benefits:

- It significantly reduces the number of memory transactions required.
- The memory layout reorganization operations are no longer needed.

Layer name	Dataset	Type of Layer	Number of outputs	Number of inputs	Filter Size	Number of Weights	Batch Size
Layer A	ImageNet	Fully-Connected Layer	4096	8192	N/A	33 Million	128
Layer B	ImageNet	Convolutional Layer	N/A	14×14	3×3×256×512	1.2 Million	64
Layer C	TIMIT	Fully-Connected Layer	1024	1024	N/A	1 Million	128

Table 7.3: Experimental setup of three representative layer configurations in our experiments.

- It saves the overhead of launching one additional kernel.
- No intermediate data is produced between these two kernels.

7.4 Experiments

In this chapter, we select the NVIDIA Tesla V100 [131] as the hardware platform to run our experiments. In terms of software, we use CUDA 10, and its associated SDKs and math libraries. We have extended the DNNMark benchmark framework [68] to implement the BCM algorithm, and leverage DNNMark tools to benchmark our implementations.

To demonstrate the effectiveness of our optimization methods, we evaluate a broad range of problem sizes using real-world models trained with the ImageNet [15] and TIMIT [195] datasets. We only consider GEMM-based layers (i.e., the convolutional and fully-connected layers) in DNNs, CNNs, TDNNs, and LSTM models [16, 17, 196].

First, we evaluate the baseline implementation of the BCM algorithm and our optimized versions versus traditional matrix multiplication (MM). For simplicity, we use three different representative layer configurations, corresponding to the fully-connected and convolutional layers in different models, using the ImageNet dataset (Layer A and B) and fully-connected layer from models using the TIMIT dataset (Layer C). To obtain these representative layer configurations, we calculate the average number of weights across all corresponding layers and select a layer configuration having the same number of weights as close as possible to the average value. For example, we find that the average number of weights across all fully-connected layers in three ImageNet trained models (AlexNet, VGGNet, and ResNet [16, 17, 19]) is approximately 30 million. As such, we select 4096 and 8192 as the number of output nodes and input nodes, generating 33 million weight parameters. Table 6.5 describes the layer configuration details of the three representative layers.

Next, we evaluate the best performing optimization method across all GEMM-based layer configurations using models trained with the ImageNet and TIMIT datasets. For the ImageNet dataset,

we select AlexNet and VGGNet-16, because they provide good coverage of a wide range of layer configurations in popular CNN models. For example, ResNet and VGGNet share the same collection of layer configurations. For the TIMIT dataset, we select the TDNN and LSTM models. For the convolutional and fully-connected layers in AlexNet and VGGNet-16, we use conv$x$ and fc$y$, where x and y are the index values for the convolutional and fully-connected layers, respectively. Note that we only present one layer configuration if multiple layers share the same configuration. For example, conv6 and conv7 of VGGNet-16 share a common layer configuration. For the fully-connected layer in TDNN and LSTM models, we use fc$x$-y to represent the layers, where the x value corresponds to the layer configuration corresponding to the number of nodes (i.e., 256, 512, 1024, and 2048), and the y index specifying the layer type (i.e., input layer, hidden layer, and output layer) [196]. To capture the execution behavior, while varying the block size, we select multiple commonly-used block sizes (16, 32, and 64).

Last, we collect a range of performance counters to analyze different optimization, including: instructions per cycle, number of instructions executed, number of memory transactions, and device occupancy. To obtain the value of the performance counters, we leverage nvprof [66], a tool that can collect the hardware performance counters on NVIDIA GPUs.

7.5 Performance Evaluation

7.5.1 Results

First, we present the performance evaluation results of our baseline implementation of the BCM algorithm, denoted as *BCM*, and then our optimized versions. We compare them against using a traditional matrix multiplication (MM) based representation (cuBLAS [156]). We label our optimized results using *O1+Sum*, *O2+Sum*, and *Kernel Fusion*, indicating which of our three optimized versions was used. *O1+Sum* implements multiplication applying the O1 optimization, along with an optimized summation. Version *O2+Sum* implements multiplication using O2 and the optimized summation. *Kernel Fusion* fuses the multiplication with O2 optimization and the summation, as described in Section 7.3.2.3.

Figure 7.5a- 7.5c show the speedup of BCM and our optimized versions over MM, across Layers A, B, and C, as described in section 7.4. From the figures, MM outperforms BCM, our baseline implementation, as described in section 7.3. The O1+Sum and O2+Sum optimizations produce substantial performance improvements over BCM. However, they cannot outperform MM.

Block Size	16	32	64
Layer A	43.3×	22.4×	11.3×
Layer B	27.3×	10.7×	4.6×
Layer C	14.7×	5.2×	2.6×

Table 7.4: Speedup of the Kernel Fusion approach over baseline BCM for Layer A, B, and C.

The Kernel Fusion approach outperforms the O1+Sum and O2+Sum optimizations. Each has a different impact on performance, depending on the problem size (a function of the number of weights, number of inputs, and batch size) when compared against MM. For Layer A, the Kernel Fusion approach can achieve a speedup of 1.12×, 2.1×, and 3.5× over MM across block sizes of 16, 32, and 64, respectively. For Layer B, the Kernel Fusion approach can also achieve a speedup of 1.06×, 1.39×, and 1.64× over MM for block sizes of 16, 32, and 64, respectively. For Layer C, the benefit of the Kernel Fusion approach is limited. From our results, we find that our proposed optimization approach can achieve better performance than MM when the problem size is sufficiently large. Although our optimization approach has limited benefits when problem sizes are small, it still outperforms the baseline implementation (BCM), significantly reducing the execution time needed for training a block-circulant DNN. Table 7.4 lists the speedup of the Kernel Fusion approach over the baseline version. From the table, the Kernel Fusion approach can achieve an average speedup of 25.7× for Layer A, 14.2× for Layer B, and 7.5× for Layer C.

We find that problem size has an impact on the performance of our optimization approach. Thus, we vary the batch size for Layers A, B, and C and present the speedup for different problem sizes in Figures 7.6a-7.6c. In this experiment, we evaluate the best performing optimization, i.e., the Kernel Fusion approach, across different block sizes, labeled as *BCM16*, *BCM32*, and *BCM64*. The baseline is MM. From the figures, we can notice that the speedup increases with increased batch size. When the batch size is sufficiently large, the trend dissipates.

In Figures 7.7a-7.7c, we present the speedup of the Kernel Fusion approach over MM, across all GEMM-based layer configurations from AlexNet and VGGNet-16, trained with the ImageNet dataset, and the TDNN and LSTM models, trained with the TIMIT dataset. From our results of using a block size of 64, we see that for AlexNet we can achieve an average speedup of 1.31× for the convolutional layers and 2.79× for the fully-connected layers. For VGGNet-16 with a block size of 64, we can achieve an average speedup of 1.33× for the convolutional layers and 3.66×

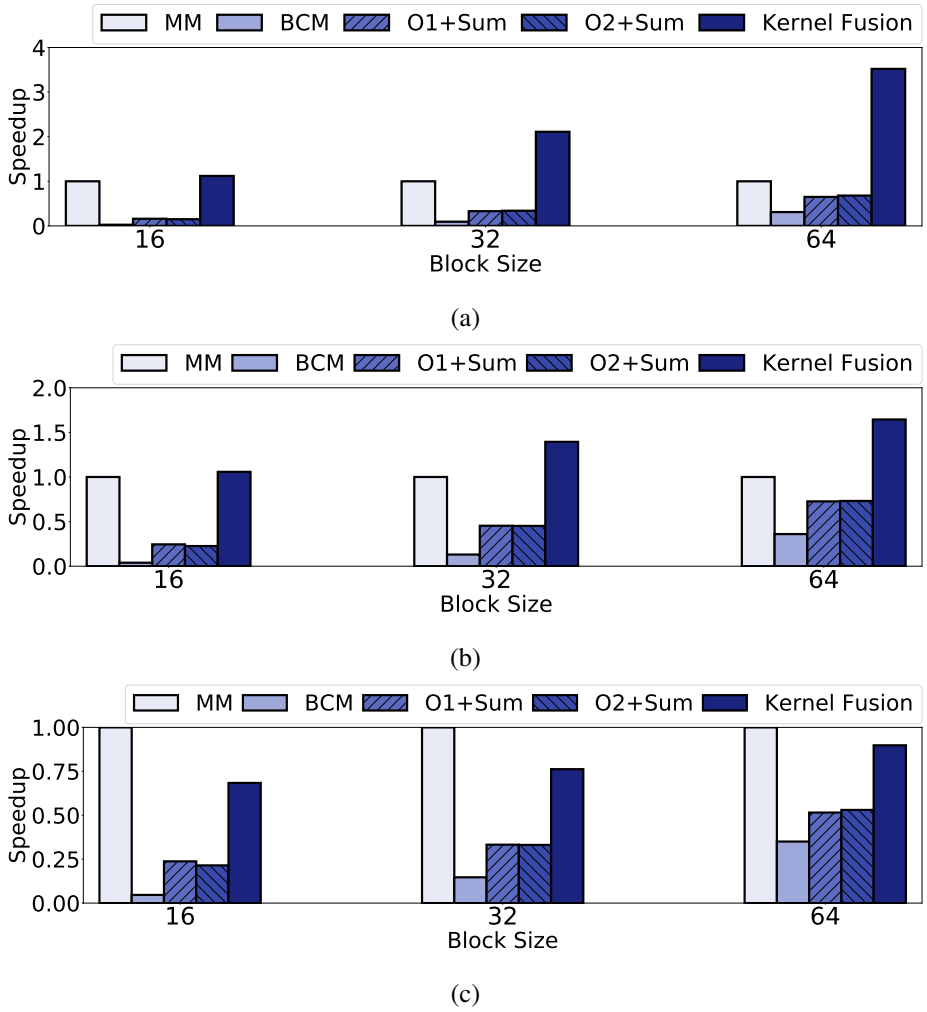


Figure 7.5: Speedup of baseline BCM version and our optimized versions, as compared to a matrix multiplication implementation for (a) Layer A, (b) Layer B, and (c) Layer C.

for the fully-connected layers. Note that two fully-connected layers in VGGNet-16 share the same layer configurations of fc2 and fc3 in AlexNet. For TDNN and LSTM, the performance gains are limited. Only when the number of weights is larger than 1.9 million can we achieve a speedup of 2.12×, given block size of 64.

7.5.2 Performance Analysis

In this section, we select a number of hardware performance counters to compare performance. We capture the number of instructions executed, the number of memory transactions, and the

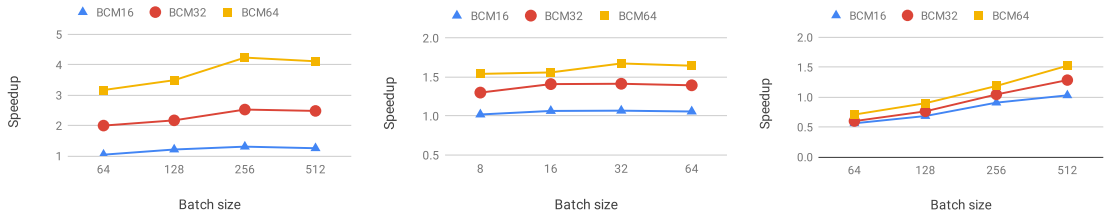


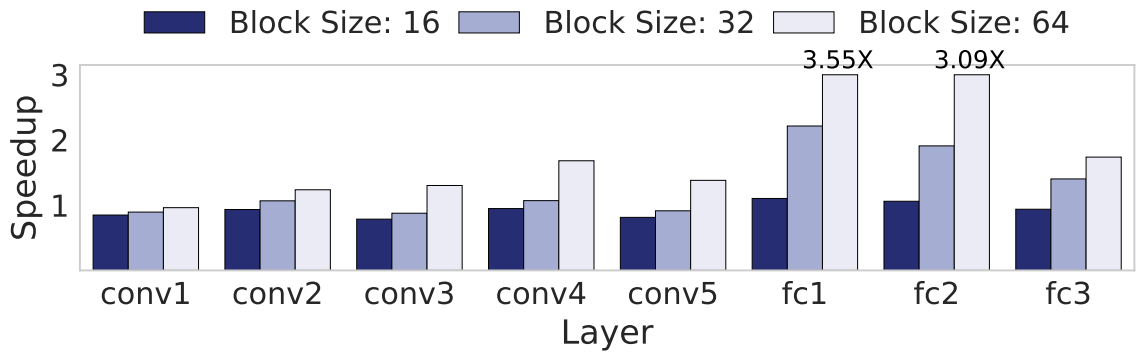
Figure 7.6: Speedup of baseline BCM version and our optimized versions, as compared to a matrix multiplication implementation for (a) Layer A, (b) Layer B, and (c) Layer C.

GPU occupancy. We use these metrics to evaluate the effectiveness of each optimization, and discuss the reasons for the resulting performance.

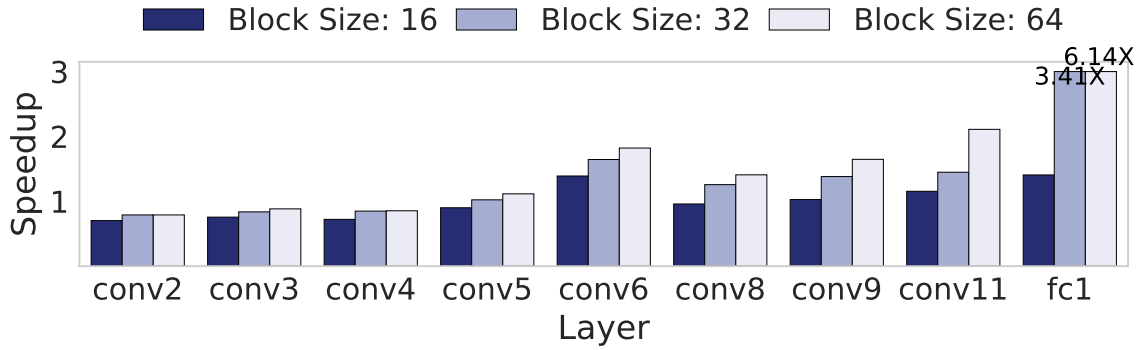
We present the implications of using the customized kernels in Figures 7.8-7.11. In Figure 7.8, we can see that the IPC is always higher for the optimized versions. For different optimizations, while each provides a different degree of performance improvement, we find that the trends in each optimization are consistent. We conclude that the number of eligible warps that can run per cycle has a strong impact on performance. The O2+Sum implementation results in the lowest number of eligible warps per cycle, as compared to the other optimized versions. We found that the resulting performance of O2+Sum is heavily impacted by the behavior of memory operations. As we explored the performance achieved in each optimized version, we found that the blockDim.x in the kernel mapping function in the multiplication kernels plays an important role. Across our implementations, a larger blockDim.x can launch more spatially local memory accesses, leading to a higher degree of memory coalescing. This translates to fewer memory transactions. O2+Sum uses k' for its blockDim.x, whereas O1+Sum and Kernel Fusion use N and $k' \times M_i$, respectively (k' is the new block size after the FFT operation, N is the new blockDim.x, as shown in Figure 7.4, and M_i is the size of the dimension over which summation is performed). Obviously, a blockDim.x of O2+Sum is smaller than the other two values.

From the same figure, we also see that the IPC of the O2+Sum optimization using a block size of 16 results in comparable performance compared to the other two schemes, This is because using a smaller block size leads to higher DR (as discussed in Section 7.3). The O2+Sum optimization is making better utilization of shared memory, thus compensating for lower memory coalescing, as described above.

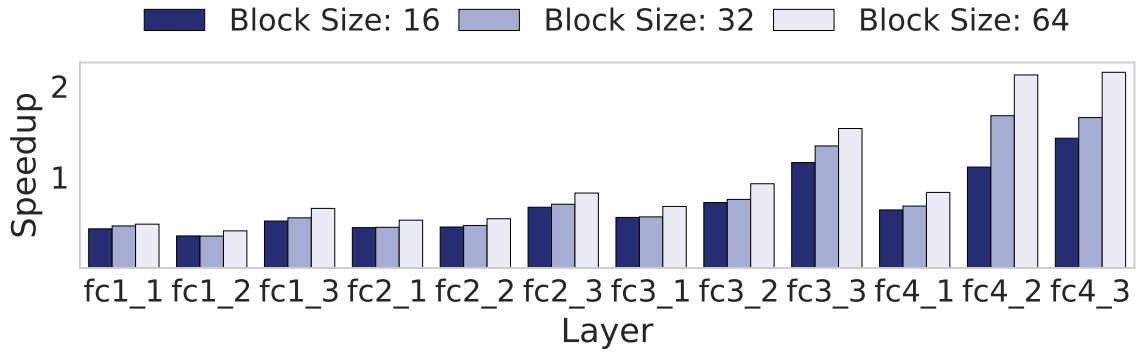
Figures 7.9a and 7.9b report two instruction count metrics, static and dynamic. The former



(a)



(b)



(c)

Figure 7.7: Speedup of the Kernel Fusion approach, as compared to a matrix multiplication implementation for all layer configurations in (a) AlexNet, (b) VGGNet-16, and (c) TDNN and LSTM.

is the number of instructions in the executable, while the latter is the number of dynamic instructions executed. Analyzing these two instruction metrics sheds light not only on the programming style

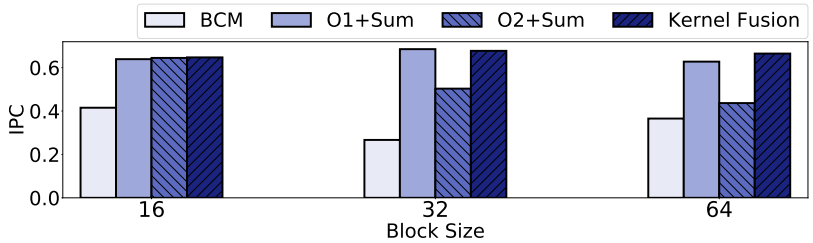
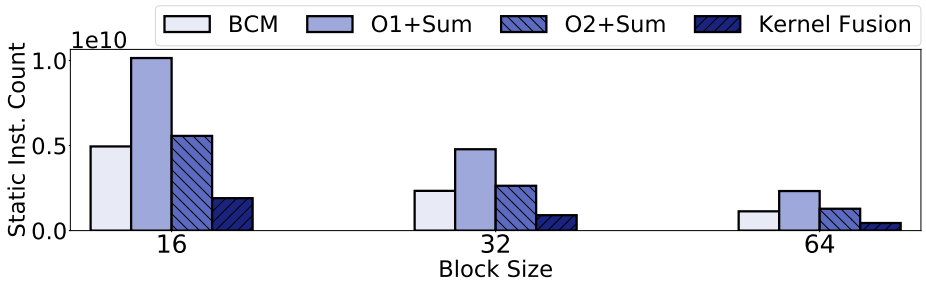
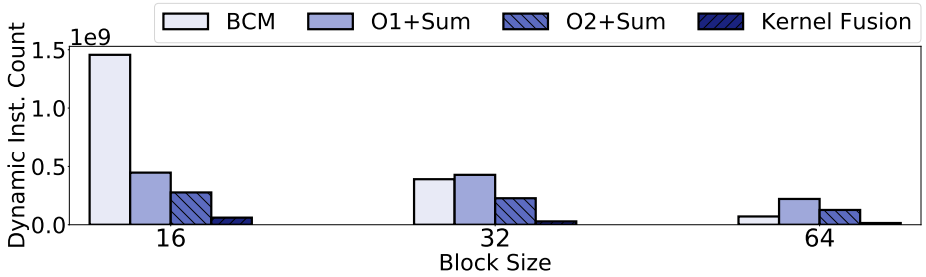


Figure 7.8: IPC of the baseline and optimized versions.



(a)



(b)

Figure 7.9: (a) Static instruction count of the baseline and optimized versions. (b) Dynamic instruction count of the baseline and optimized versions.

used in each optimization, but also on the execution efficiency of each optimization. From our results, we can see that the baseline version has a low static instruction count, given that the implementation is straightforward. However, its high dynamic instruction count shown in Figure 7.9b indicates that the resulting execution is highly inefficient. The optimized versions result in better execution efficiency than the baseline.

To better quantify the execution efficiency, we use a ratio R of these two instruction counts ($N_{static}/N_{dynamic}$). The ratio R reflects the execution efficiency on a GPU. As shown in table 7.5. Kernel Fusion has the largest R, as it removes a large number of memory operations, therefore is

Table 7.5: Ratio of the number of static and dynamic instructions.

BCM	O1+Sum	O2+Sum	Kernel Fusion
3.4	22.7	20.1	31.3

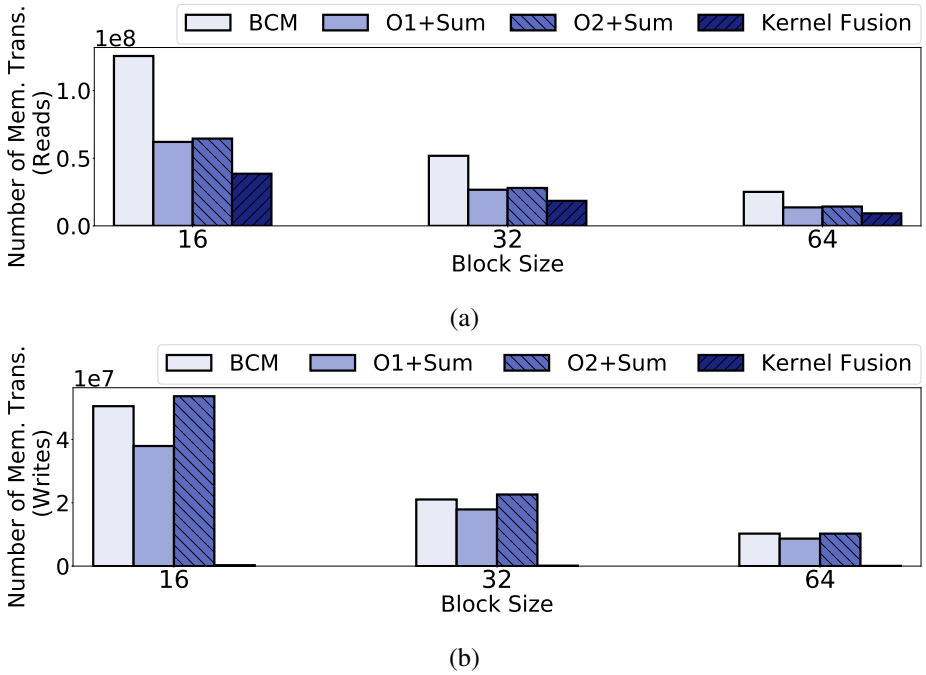


Figure 7.10: (a) Number of memory transactions (Reads) for the baseline and optimized versions. (b) Number of memory transactions (Writes) for the baseline and optimized versions.

less memory bound than the other codes. The O1+Sum optimization results in a better R value than O2+Sum, but still has a larger dynamic instruction count, as shown in Figure 7.9b. The O1+Sum executable has a much higher static instruction count. The implementation of O1+Sum requires more integer operations to calculate the data indices and more branch logic for avoiding out-of-bounds data indexing.

Next, we present the number of memory transactions in Figures 7.10a and 7.10b, reporting the number of memory reads and writes, respectively. We can see that the Kernel Fusion optimization significantly reduces the number of memory transactions for both reads and writes. We can also see that the number of memory transactions drops as the block size increases. This is because, as the block size grows, there are fewer circulant matrix blocks. As such, the number of inter-block memory accesses is reduced, leading to a more regular memory access pattern, and hence, fewer

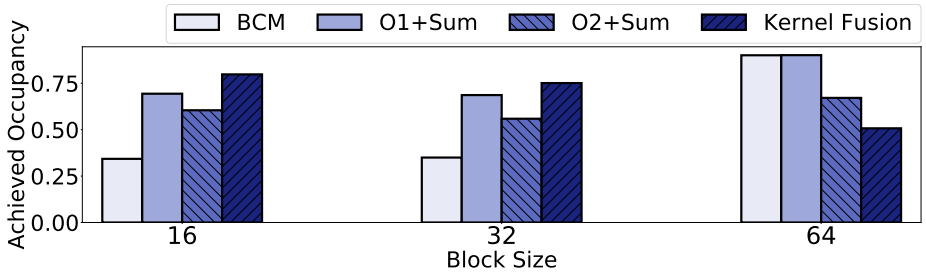


Figure 7.11: The occupancy achieved for the baseline and optimized versions.

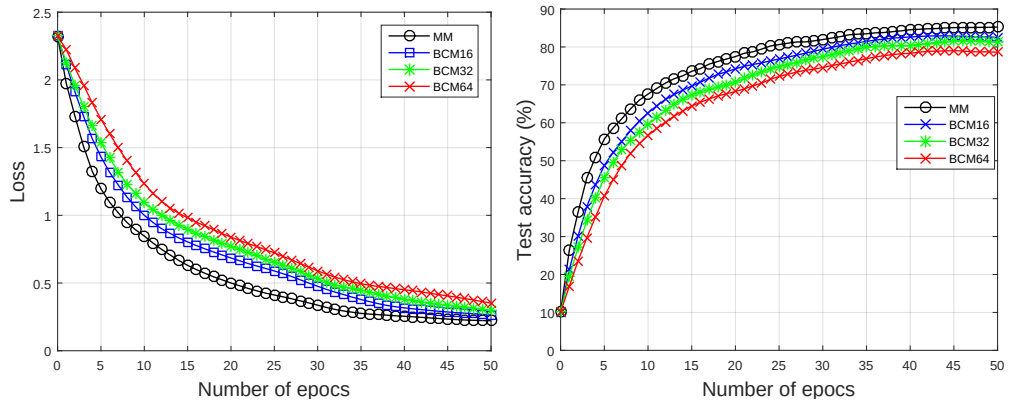
memory transactions.

Figure 7.11 shows the effectiveness of each optimization in terms of GPU occupancy, across different block sizes (16 and 32). However, as the block size k grows to 64, Kernel Fusion results in a lower occupancy than that of other implementations. This is because its thread block size decreases monotonically as a function of k, i.e., $512 + \frac{1024}{k}$. Unlike Kernel Fusion, the O2+Sum optimization achieves an unexpected increase in occupancy when using a block size of 64, even though it includes similar optimizations as used in Kernel Fusion. This is due to the impact of increased occupancy of the summation kernel. The multiplication kernel in O2+Sum actually has reduced occupancy, for the same reason as was experienced using Kernel Fusion.

7.5.3 Compression Rate V.S. Training Performance

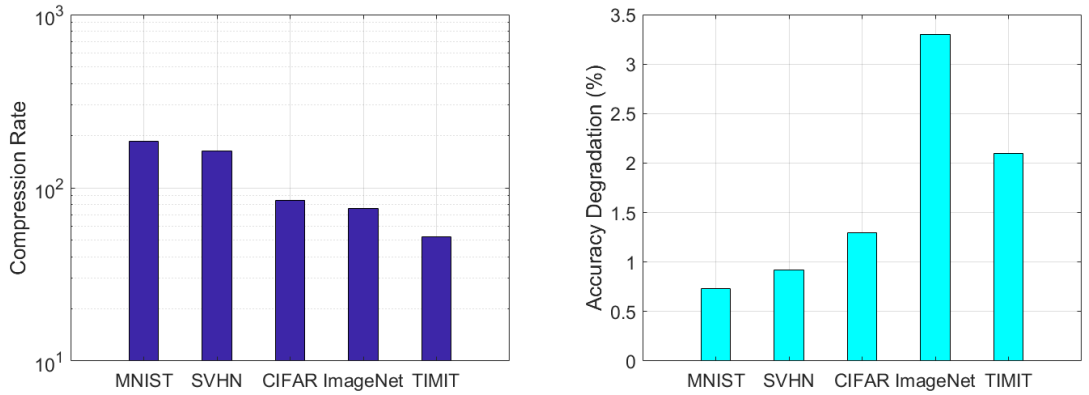
To begin, we use the BCM algorithm to train a DNN model that has 4 convolutional layers, 3 fully connected layers and a softmax layer, with the CIFAR-10 dataset [179]. We use block sizes of 16 (BCM16), 32 (BCM32) and 64 (BCM64). Figures 7.12a and 7.12b compare the training convergence rate and classification accuracy for GEMM versus our BCM algorithm. From the figures, we can see that the BCM algorithm obtains a similar convergence rate as compared with the GEMM approach. We also find that the BCM algorithm achieves a similar classification accuracy as using GEMM. We find that the accuracy degradation is closely correlated with the training convergence rate. The less the accuracy degrades, the less training convergence suffers.

Next, to demonstrate the effectiveness of training using the BCM algorithm, in Figure 7.13a we show the compression rate of models trained with different datasets, including MNIST [197], SHVN [198], CIFAR-10 [179], ImageNet [15] and TIMIT [195]. We can achieve a significant reduction in model size, as shown in the figure. We also report the accuracy degradation of the same models in Figure 7.13b. For each model, the degradation is smaller than 4%, as compared with the



(a) (b)

Figure 7.12: (a) Training convergence rate on CIFAR-10 dataset. (b) Test accuracy on CIFAR-10 dataset.



(a) (b)

Figure 7.13: (a) Compression rate and (b) Accuracy degradation of models trained with datasets, including MNIST, SHVN, CIFAR-10, ImageNet, and TIMIT.

original models.

7.5.4 Multi-GPU DNN Training

Multi-GPU DNN training has been widely deployed to shorten the training time[120]. In multi-GPU training, the same DNN model is duplicated on distributed GPU nodes and the forward and backward propagation are executed independently with different batches of input data. Only the

gradients need to be collected from the different nodes and aggregated to update the weights.

Using BCM also benefits multi-GPU training as it can significantly reduce the amount of data transferred during the weight update stage. We use Caffe[59] to conduct an experiment involving two PCIe-connected NVIDIA V100 GPUs and perform training for a simple model. We replace one regular linear transformation layer with a block circulant layer using the Kernel Fusion optimization. When the block size is 64, we achieve a 6.25X speedup over using a matrix multiplication.

Chapter 8

Conclusion

Deep Neural Networks (DNNs) have become the main focal point of AI research, inspiring a broad range of applications including, but not limited to, image/speech recognition, object localization/detection, and natural language processing. A modern DNN architecture can contain millions of parameters and hundreds of layers, placing massive amounts of computational demand on existing hardware platforms. Of many popular platforms, GPUs have been used effectively to efficiently train a DNN, enabling this class of algorithms to reach unprecedented popularity. However, there are many challenges to run a DNN on GPUs efficiently. First, there is a void of tools available to measure and tune the performance of DNNs on a GPU. Second, there is limited prior work regarding the execution of this type of workload on GPUs. Lastly, existing methods that exploit sparsity and model compression, have many limitations when run on a GPU. Directly applying those techniques can lead to significant inefficiencies, potentially degrading performance. Given these challenges, we have focusing on tool development, enabling us to explore new methods and optimizations to accelerate DNNs on a GPU.

In chapter 4, we develop a new benchmark suite named DNNMark, a deep neural network benchmark suite designed to provide a configurable, extensible, and flexible method for profiling computation of individual DNN primitives or complete DNN models. DNNMark supports both general and algorithm-specific configurability and creates the possibility of extending current DNN primitives to complex models as new benchmarks. From our evaluation results, our metrics can help illustrate specific compute patterns associated with DNNs, providing an invaluable guide for future research.

Next, in chapter 5, we characterize the demands placed on a GPU microarchitecture while running a commonly used CNN model (AlexNet). We consider performance on a layer-by-layer basis.

We carefully select metrics that can characterize the execution behavior of each layer in the model and identify the major limiting factors for each layer. From our evaluation, we find that the characteristics of each layer vary significantly due to the distinct type of operations performed. Based on the microarchitectural demands imposed by each layer, we identify the significant bottlenecks present in both an entire CNN model, and each layer, and suggest several optimization approaches that can improve the performance with only minor changes and overhead. Motivated by our characterization results and previous studies, we explore both software and hardware optimization methods, involving approaches, such as sparsity exploitation and weight compression.

In chapter 6, we present Spartan, a framework leveraging activation sparsity to accelerate DNN training on a GPU. Our work presents the following contributions:

- We propose a novel sparsity monitor that intelligently acquires and tracks activation sparsity with negligible overhead. Using our monitor, we can significantly reduce sparsity profiling overhead by 52.5×, on average. We adopt a periodical monitoring technique, whose behavior is regulated by two algorithms: i) a Dynamic Period Adjustment Algorithm (DPAA), and ii) a Sparsity Stability Detecting Algorithm (SSDA). The former dynamically adjusts the monitoring period, while the latter detects the sparsity stability level.

- We propose a novel sparse format ELLPACK-DIB (Data Index Bundling) based on ELLPACK-R [113], and design a customized tile-based sparse GEMM algorithm that uses this format.

- We propose a novel compaction engine located between the L2 cache and main memory of the GPU, enabling dynamic compaction/conversion. It serves as a near memory processing unit, responsible for compacting sparse data into the ELLPACK-DIB format during kernel execution. The compaction engine consists of three major components: i) a Sparsity Information Block (SIB), ii) a Compacting Processor, and iii) a Prefetch Buffer. We further customize the GEMM algorithm to utilize our enhanced GPU architecture incorporating the Spartan compaction engine.

- We evaluate our sparsity monitor during the training process with five commonly-used DNN models, using both CIFAR-10 and ImageNet datasets. We find that the average overhead introduced by profiling is only 1.7%. We evaluate the compaction engine using our customized sparse GEMM algorithm. For convolutional layers, we can achieve an average speedup of 3.4× for AlexNet, 2.14× for VGGNet-16, and 2.02× for ResNet-18, when training on the ImageNet dataset.

In chapter 7, we present our work accelerating DNN training using Block Circulant Matrices. Our work makes the following contributions:

- We extend the BCM algorithm for batched DNN training on a GPU.
- We propose a set of optimizations for the BCM algorithm when running DNN training, reducing the number of redundant computations during both forward and backward propagation, taking advantage of the mathematical properties of an FFT/IFFT, and modifying selected steps in the algorithm.
- We discuss a series of GPU kernel tuning principles, considering three factors that influence performance: i) device occupancy, ii) data reuse, and iii) memory coalescing for both reads and writes. We define two new metrics that can characterize data reuse and memory coalescing patterns present in BCM-based DNNs. Equipped with these metrics, as well a measure of device occupancy, we can guide our optimizations to accelerate DNN execution effectively.
- We evaluate the execution of both forward and backward propagation on an NVIDIA Tesla V100, employing performance counters to profile the resulting execution. We report on the instruction per cycle (IPC), static/dynamic instruction count, the memory intensity (the total number of memory transactions issued across all GPU kernels), and device occupancy.
- We show that by applying our optimizations with BCM, we can achieve an average speedup of 1.31× and 2.79× for the convolutional layers and fully-connected layers of AlexNet, respectively, and a speedup of 1.33× and 3.66× for the convolutional layers and fully-connected layers of VGGNet-16, respectively. Besides, when using BCM during the DNN training process, the model experiences a minimal loss in terms of both convergence rate and inference accuracy. We also show that there can be significant benefits when using BCM to reduce the amount of communication required during a multi-GPU training using two NVIDIA Tesla V100's.

8.1 Future Work

- **Characterization:** We can expand our proposed characterization methodology to more models. Currently, we only characterized the CNN model. It is also essential to understand the execution behavior of a different type of neural network models, such as Long Short Term Memory

(LSTM) [187], Generative Adversarial Networks (GAN) [199], and Graph Neural Networks (GNN) [200], on GPUs. Also, we only characterized DNN training on desktop-grade and server-grade GPUs. It is also critical to characterize DNN inference on edge computing devices, involving a different class of models and platforms.

- **Spartan:** We can perform a broader design space exploration for Spartan, tuning the parameters and mechanisms of the compaction engine. Some potential next steps can be: 1) Change the L2 memory space allocated to the compaction engine, observing the difference of performance and memory footprints, 2) Design different CMI replacement mechanisms, and analyze the pro and con. Besides, we can study the impact of the sparse pattern on performance. Currently, we only consider compact sparse data in a fine-grained manner. Previous studies suggest a coarse-grained sparse pattern (i.e., block sparsity) can lead to better performance [116, 106]. Some potential next steps can be: 1) Characterize the block sparsity during the DNN training, and 2) Optimize the design of the compaction engine, compacting the sparse data in a coarse-grained fashion.

- **Weight compression:** We can study the possibility of mapping multiple stages of the BCM algorithm to different GPUs, exploring the benefit of the model parallelism [201]. When using the BCM algorithm, multiple GPU kernels are needed, leading to additional memory usage. We can optimize memory usage when deploying the BCM algorithm, especially since memory usage becomes a growing concern as DNNs grow deeper and wider. Other than BCMs, many other structured matrices can potentially be exploited for DNNs. For example, Permuted Diagonal Matrices [202]. We can explore the potential benefits of other structured matrices on GPU, following the kernel tuning principles we discussed.

Bibliography

[1] P. Langley and H. A. Simon, "Applications of machine learning and rule induction," *Commun. ACM*, vol. 38, no. 11, pp. 54–64, Nov. 1995. [Online]. Available: http://doi.acm.org.ezproxy.neu.edu/10.1145/219717.219768

[2] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern Classification (2Nd Edition)*. Wiley-Interscience, 2000.

[3] P. Flach, *Machine Learning: The Art and Science of Algorithms That Make Sense of Data*. New York, NY, USA: Cambridge University Press, 2012.

[4] D. Zhang and J. J. P. Tsai, *Advances in Machine Learning Applications in Software Engineering*. Hershey, PA, USA: IGI Global, 2007.

[5] S. Dong, Z. Feric, X. Li, S. M. Rahman, G. Li, C. Wu, A. Z. Gu, J. Dy, D. Kaeli, J. Meeker, I. Y. Padilla, J. Cordero, C. V. Vega, Z. Rosario, and A. Alshawabkeh, "A hybrid approach to identifying key factors in environmental health studies," in *2018 IEEE International Conference on Big Data (Big Data)*, Dec 2018, pp. 2855–2862.

[6] M. W. Libbrecht and W. S. Noble, "Machine learning applications in genetics and genomics," *Nature Reviews Genetics*, vol. 16, pp. 321–332, 2015.

[7] J. A. Cruz and D. S. Wishart, "Applications of machine learning in cancer prediction and prognosis," *Cancer Informatics*, vol. 2, pp. 59 – 77, 2006.

[8] X. Chen and X. Lin, "Big data deep learning: Challenges and perspectives," *IEEE Access*, vol. 2, pp. 514–525, 2014.

[9] J. M. Stanton, "Galton, pearson, and the peas: A brief history of linear regression for statistics instructors," *Journal of Statistics Education: An international journal on the teaching and learning of statistics*, vol. 9, no. 3, pp. 3–3, 2001.

[10] S. Hussain, M. Hashmani, M. Moinuddin, M. Yoshida, and H. Kanjo, "Image retrieval based on color and texture feature using artificial neural network," in *Emerging Trends and Applications in Information Communication Technologies*, B. S. Chowdhry, F. K. Shaikh, D. M. A. Hussain, and M. A. Uqaili, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 501–511.

[11] K. Pearson, "On lines and planes of closest fit to systems of points in space," *Philosophical Magazine*, vol. 2, pp. 559–572, 1901.

[12] R. S. Choras, "Feature extraction for cbir and biometrics applications," in *Proceedings of the 7th Conference on 7th WSEAS International Conference on Applied Computer Science - Volume 7*, ser. ACS'07. Stevens Point, Wisconsin, USA: World Scientific and Engineering Academy and Society (WSEAS), 2007, pp. 1–9. [Online]. Available: http://dl.acm.org/citation.cfm?id=1348171.1348172

[13] J. Yue, Z. Li, L. Liu, and Z. Fu, "Content-based image retrieval using color and texture fused features," *Math. Comput. Model.*, vol. 54, no. 3-4, pp. 1121–1127, Aug. 2011. [Online]. Available: http://dx.doi.org/10.1016/j.mcm.2010.11.044

[14] S. Hussain, M. Hashmani, M. Moinuddin, M. Yoshida, and H. Kanjo, "Image retrieval based on color and texture feature using artificial neural network," in *Emerging Trends and Applications in Information Communication Technologies*, B. S. Chowdhry, F. K. Shaikh, D. M. A. Hussain, and M. A. Uqaili, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 501–511.

[15] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009.

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105.

[17] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations*, 2015.

[18] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015, pp. 1–9.

[19] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016, pp. 770–778.

[20] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, "Deepface: Closing the gap to human-level performance in face verification," in *2014 IEEE Conference on Computer Vision and Pattern Recognition*, June 2014, pp. 1701–1708.

[21] J. Tang, D. Sun, S. Liu, and J. Gaudiot, "Enabling deep learning on iot devices," *Computer*, vol. 50, no. 10, pp. 92–96, 2017.

[22] B. Huval, T. Wang, S. Tandon, J. Kiske, W. Song, J. Pazhayampallil, M. Andriluka, P. Rajpurkar, T. Migimatsu, R. Cheng-Yue, F. A. Mujica, A. Coates, and A. Y. Ng, "An empirical evaluation of deep learning on highway driving," *CoRR*, vol. abs/1504.01716, 2015. [Online]. Available: http://arxiv.org/abs/1504.01716

[23] G. J. S. Litjens, T. Kooi, B. E. Bejnordi, A. A. A. Setio, F. Ciompi, M. Ghafoorian, J. A. W. M. van der Laak, B. van Ginneken, and C. I. Snchez, "A survey on deep learning in medical image analysis." *Medical Image Analysis*, vol. 42, pp. 60–88, 2017. [Online]. Available: http://dblp.uni-trier.de/db/journals/mia/mia42.html#LitjensKBSCGLGS17

[24] V. Kpuska and G. Bohouta, "Next-generation of virtual personal assistants (microsoft cortana, apple siri, amazon alexa and google home)," in *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*, Jan 2018, pp. 99–103.

[25] A. Roy, J. Sun, R. Mahoney, L. Alonzi, S. Adams, and P. Beling, "Deep learning detecting fraud in credit card transactions," in *2018 Systems and Information Engineering Design Symposium (SIEDS)*, April 2018, pp. 129–134.

[26] H. Wang, N. Wang, and D.-Y. Yeung, "Collaborative deep learning for recommender systems," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '15. New York, NY, USA: ACM, 2015, pp. 1235–1244. [Online]. Available: http://doi.acm.org/10.1145/2783258.2783273

[27] F. F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain." *Psychological review*, vol. 65 6, pp. 386–408, 1958.

[28] W. Mcculloch and W. Pitts, "A logical calculus of ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, vol. 5, pp. 127–147, 1943.

[29] J. Schaeffer, *One Jump Ahead: Challenging Human Supremacy in Checkers*, ser. Copernicus Series. Springer, 1997. [Online]. Available: https://books.google.com/books?id=SLvpkTVhsIsC

[30] A. Newell and H. Simon, "The logic theory machine–a complex information processing system," *IRE Transactions on Information Theory*, vol. 2, no. 3, pp. 61–79, Sep. 1956.

[31] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. [Online]. Available: http://dx.doi.org/10.1162/neco.1997.9.8.1735

[32] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the National Academy of Sciences*, vol. 79, no. 8, pp. 2554–2558, 1982. [Online]. Available: https://www.pnas.org/content/79/8/2554

[33] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1," D. E. Rumelhart, J. L. McClelland, and C. PDP Research Group, Eds. Cambridge, MA, USA: MIT Press, 1986, ch. Learning Internal Representations by Error Propagation, pp. 318–362. [Online]. Available: http://dl.acm.org/citation.cfm?id=104279.104293

[34] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998.

[35] B. Reagen, R. Adolf, and P. Whatmough, *Deep Learning for Computer Architects*. Morgan & Claypool Publishers, 2017.

[36] G. Stockman and L. G. Shapiro, *Computer Vision*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.

[37] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proceedings 22nd Annual International Symposium on Computer Architecture*, June 1995, pp. 392–403.

[38] C. Lomont, "Introduction to intel advanced vector extensions. intel white paper," 2011.

[39] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, p. 112, Jun. 2017. [Online]. Available: https://doi.org/10.1145/3140659.3080246

[40] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, and J. Cong, "Energy-efficient cnn implementation on a deeply pipelined fpga cluster," in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, ser. ISLPED '16. New York, NY, USA: ACM, 2016, pp. 326–331. [Online]. Available: http://doi.acm.org/10.1145/2934583.2934644

[41] Google, "Tensorflow alexnet benchmark." [Online]. Available: https://www.leadergpu.com/articles/428-tensorflow-alexnet-benchmark.

[42] T. Geng, T. Wang, A. Sanaullah, C. Yang, R. Xu, R. Patel, and M. C. Herbordt, "Fpdeep: Acceleration and load balancing of cnn training on fpga clusters," *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 81–84, 2018.

[43] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 243–254.

[44] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. B. J. Dally, "Ese: Efficient speech recognition engine with sparse lstm on fpga," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate*

Arrays, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 75–84. [Online]. Available: http://doi.acm.org/10.1145/3020078.3021745

[45] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen, "Cambricon-s: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2018, pp. 15–28.

[46] C. Ding, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, Y. Bai, G. Yuan, X. Ma, Y. Zhang, J. Tang, Q. Qiu, X. Lin, and B. Yuan, "Circnn: Accelerating and compressing deep neural networks using block-circulant weight matrices," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 395–408. [Online]. Available: http://doi.acm.org/10.1145/3123939.3124552

[47] V. Podlozhnyuk, "Image convolution with cuda," 2013.

[48] ——, "Fft-based 2d convolution," 2007.

[49] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," *arXiv preprint arXiv:1509.09308*, 2015.

[50] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, p. 436444, 2015.

[51] R. A. Cohen, *Lateral Inhibition*. Springer New York, 2011.

[52] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015. [Online]. Available: http://arxiv.org/abs/1502.03167

[53] G. B. Orr and K.-R. Mueller, Eds., *Neural Networks : Tricks of the Trade*, ser. Lecture Notes in Computer Science. Springer, 1998, vol. 1524.

[54] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ACM Sigplan Notices*, vol. 49, no. 4, 2014, pp. 269–284.

[55] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, "Going deeper with embedded fpga platform for convolutional neural network," in *FPGA*. ACM, 2016, pp. 26–35.

[56] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *MICRO*. IEEE Computer Society, 2014, pp. 609–622.

[57] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernndez-Lobato, G. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 267–278.

[58] S. Dong, X. Gong, Y. Sun, T. Baruah, and D. Kaeli, "Characterizing the microarchitectural implications of a convolutional neural network (cnn) execution on gpus," in *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18. New York, NY, USA: ACM, 2018, pp. 96–106.

[59] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," 2014.

[60] J. Gao, W. Ji, Z. Tan, and Y. Zhao, "A systematic survey of general sparse matrix-matrix multiplication," 2020.

[61] NVIDIA, "cusparse," 2019. [Online]. Available: https://docs.nvidia.com/cuda/cusparse/index.html

[62] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," 2015.

[63] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural network," in *NIPS*, 2015, pp. 1135–1143.

[64] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko, "Gist: Efficient data encoding for deep neural network training," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018, pp. 776–789.

[65] Y. Sun, N. B. Agostini, S. Dong, and D. Kaeli, "Summarizing cpu and gpu design trends with product data," 2019.

[66] N. Corporation, "CUDA C Programming Guide," 2014.

[67] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. Mccardwell, A. Villegas, and D. Kaeli, "Hetero-mark, a benchmark suite for cpu-gpu collaborative computing," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, Sep. 2016, pp. 1–10.

[68] S. Dong and D. Kaeli, "Dnnmark: A deep neural network benchmark suite for gpus," in *Proceedings of the General Purpose GPUs*, ser. GPGPU-10. New York, NY, USA: ACM, 2017, pp. 63–72. [Online]. Available: http://doi.acm.org/10.1145/3038228.3038239

[69] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 44–54, 2009.

[70] J. A. Stratton, C. I. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. Liu, and W. mei W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," 2012.

[71] J. Gmez-Luna, I. E. Hajj, L. Chang, V. Garca-Floreszx, S. G. de Gonzalo, T. B. Jablin, A. J. Pea, and W. Hwu, "Chai: Collaborative heterogeneous applications for integrated-architectures," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2017, pp. 43–54.

[72] Y. Ukidave, F. N. Paravecino, L. Yu, C. Kalra, A. Momeni, Z. Chen, N. Materise, B. Daley, P. Mistry, and D. Kaeli, "Nupar: A benchmark suite for modern gpu architectures," *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, pp. 253–264, 2015.

[73] A. Li, S. L. Song, J. Chen, X. Liu, N. Tallent, and K. Barker, "Tartan: Evaluating modern gpu interconnect via a multi-gpu benchmark suite," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, Sep. 2018, pp. 191–202.

[74] Y. Sun, T. Baruah, S. A. Mojumder, S. Dong, R. Ubal, X. Gong, S. Treadway, Y. Bao, V. Zhao, J. L. Abelln, J. Kim, A. Joshi, and D. Kaeli, "Mgsim + mgmark: A framework for multi-gpu system research," 2018.

[75] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[76] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. [Online]. Available: http://tensorflow.org/

[77] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[78] S. Narang, "Deepbench," 2016.

[79] S. Verma, Q. Wu, B. Hanindhito, G. Jha, E. B. John, R. Radhakrishnan, and L. K. John, "Demystifying the mlperf benchmark suite," 2019.

[80] H. Zhu, M. Akrout, B. Zheng, A. Pelegris, A. Jayarajan, A. Phanishayee, B. Schroeder, and G. Pekhimenko, "Benchmarking and analyzing deep neural network training," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, Sep. 2018, pp. 88–100.

[81] B. Hu and C. J. Rossbach, "Mirovia: A benchmarking suite for modern heterogeneous computing," 2019.

[82] A. Karki, C. Palangotu Keshava, S. Mysore Shivakumar, J. Skow, G. Madhukeshwar Hegde, and H. Jeon, "Tango: A deep neural network benchmark suite for various accelerators," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2019, pp. 137–138.

[83] W. Gao, C. Luo, L. Wang, X. Xiong, J. Chen, T. Hao, Z. Jiang, F. Fan, M. Du, Y. Huang, F. Zhang, X. Wen, C. Zheng, X. He, J. Dai, H. Ye, Z. Cao, Z. Jia, K. Zhan, H. Tang, D. Zheng, B. Xie, W. Li, X. Wang, and J. Zhan, "Aibench: Towards scalable and comprehensive datacenter ai benchmarking," in *Benchmarking, Measuring, and Optimizing*, C. Zheng and J. Zhan, Eds. Cham: Springer International Publishing, 2019, pp. 3–9.

[84] S. Shi, Q. Wang, P. Xu, and X. Chu, "Benchmarking State-of-the-Art Deep Learning Software Tools," *CoRR*, vol. abs/1608.07249, 2016. [Online]. Available: http://arxiv.org/abs/1608.07249

[85] H. Kim, H. Nam, W. Jung, and J. Lee, "Performance analysis of CNN frameworks for GPUs," *Performance Analysis of Systems and Software (ISPASS)*, 2017.

[86] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design," *Microarchitecture (MICRO)*, 2016.

[87] S. A. Mojumder, M. S. Louis, Y. Sun, A. K. Ziabari, J. L. Abelln, J. Kim, D. Kaeli, and A. Joshi, "Profiling dnn workloads on a volta-based dgx-1 system," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, Sep. 2018, pp. 122–133.

[88] Y. Sun, S. Mukherjee, T. Baruah, S. Dong, J. Gutierrez, P. Mohan, and D. Kaeli, "Evaluating performance tradeoffs on the radeon open compute platform," in *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2018, pp. 209–218.

[89] A. Karki, C. P. Keshava, S. M. Shivakumar, J. Skow, G. M. Hegde, and H. Jeon, "Detailed characterization of deep neural networks on gpus and fpgas," in *Proceedings of the 12th Workshop on General Purpose Processing Using GPUs*, ser. GPGPU 19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1221. [Online]. Available: https://doi.org/10.1145/3300053.3319418

[90] C. Ding, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, Y. Bai, G. Yuan, and X. Lin, "C ir cnn: accelerating and compressing deep neural networks using block-circulant weight matrices," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017, pp. 395–408.

[91] Z. Yao, S. Cao, W. Xiao, C. Zhang, and L. Nie, "Balanced sparsity for efficient DNN inference on GPU," *CoRR*, vol. abs/1811.00206, 2018. [Online]. Available: http://arxiv.org/abs/1811.00206

[92] M. Mathieu, M. Henaff, and Y. LeCun, "Fast training of convolutional networks through ffts," *CoRR*, 2014. [Online]. Available: https://arxiv.org/abs/1312.5851

[93] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[94] G. Castellano, A. M. Fanelli, and M. Pelillo, "An iterative pruning algorithm for feedforward neural networks," *IEEE transactions on neural networks*, vol. 8 3, pp. 519–31, 1997.

[95] S. Anwar, K. Hwang, and W. Sung, "Structured pruning of deep convolutional neural networks," *J. Emerg. Technol. Comput. Syst.*, vol. 13, no. 3, Feb. 2017. [Online]. Available: https://doi.org/10.1145/3005348

[96] Q. Huang, S. K. Zhou, S. You, and U. Neumann, "Learning to prune filters in convolutional neural networks," *CoRR*, vol. abs/1801.07365, 2018. [Online]. Available: http://arxiv.org/abs/1801.07365

[97] T. Zhang, S. Ye, K. Zhang, J. Tang, W. Wen, M. Fardad, and Y. Wang, "A systematic dnn weight pruning framework using alternating direction method of multipliers," in *The European Conference on Computer Vision (ECCV)*, September 2018.

[98] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *ICML*, 2016, pp. 2849–2858.

[99] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, "Quantized convolutional neural networks for mobile devices," in *CVPR*, 2016.

[100] M. Jaderberg, A. Vedaldi, and A. Zisserman, "Speeding up convolutional neural networks with low rank expansions," 2014.

[101] C. Tai, T. Xiao, X. Wang, and W. E, "Convolutional neural networks with low-rank regularization," 2015.

[102] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, ser. NIPS16. Red Hook, NY, USA: Curran Associates Inc., 2016, p. 20822090.

[103] Baoyuan Liu, Min Wang, H. Foroosh, M. Tappen, and M. Penksy, "Sparse convolutional neural networks," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015, pp. 806–814.

[104] S. Narang, E. Undersander, and G. F. Diamos, "Block-sparse recurrent neural networks," *CoRR*, vol. abs/1711.02782, 2017. [Online]. Available: http://arxiv.org/abs/1711.02782

[105] S. Han, J. Pool, S. Narang, H. Mao, E. Gong, S. Tang, E. Elsen, P. Vajda, M. Paluri, J. Tran, B. Catanzaro, and W. J. Dally, "Dsd: Dense-sparse-dense training for deep neural networks," in *International Conference on Learning Representations (ICLR)*, 2017.

[106] M. Ren, A. Pokrovsky, B. Yang, and R. Urtasun, "Sbnet: Sparse blocks network for fast inference," *CoRR*, vol. abs/1801.02108, 2018. [Online]. Available: http://arxiv.org/abs/1801.02108

[107] P. Judd, A. D. Lascorz, S. Sharify, and A. Moshovos, "Cnvlutin2: Ineffectual-activation-and-weight-free deep neural network computing," *CoRR*, vol. abs/1705.00125, 2017. [Online]. Available: http://arxiv.org/abs/1705.00125

[108] S. Shi and X. Chu, "Speeding up convolutional neural networks by exploiting the sparsity of rectifier units," *CoRR*, vol. abs/1704.07724, 2017. [Online]. Available: http://arxiv.org/abs/1704.07724

[109] GPUOpen, "clsparse," 2016. [Online]. Available: https://gpuopen.com/compute-product/clsparse/

[110] AMD, "rocsparse," 2019. [Online]. Available: https://github.com/ROCmSoftwarePlatform/rocSPARSE

[111] W. Liu and B. Vinter, "Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS 15. New York, NY, USA: Association for Computing Machinery, 2015, p. 339350. [Online]. Available: https://doi-org.ezproxy.neu.edu/10.1145/2751205.2751209

[112] M. Steinberger, R. Zayer, and H.-P. Seidel, "Globally homogeneous, locally adaptive sparse matrix-vector multiplication on the gpu," in *Proceedings of the International Conference on Supercomputing*, ser. ICS 17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3079079.3079086

[113] F. Vzquez, G. Ortega, J. J. Fernndez, and E. M. Garzn, "Improving the performance of the sparse matrix vector product with gpus," in *2010 10th IEEE International Conference on Computer and Information Technology*, June 2010, pp. 1146–1151.

[114] W. Liu and B. Vinter, "An efficient gpu general sparse matrix-matrix multiplication for irregular data," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 370–381.

[115] S. Dalton, L. Olson, and N. Bell, "Optimizing sparse matrixmatrix multiplication for the gpu," *ACM Trans. Math. Softw.*, vol. 41, no. 4, Oct. 2015. [Online]. Available: https://doi.org/10.1145/2699470

[116] A. R. Scott Gray and D. P. Kingma, "Gpu kernels for block-sparse weights," *OpenAI*, 2017.

[117] C. Yang, A. Buluc, and J. D. Owens, "Design principles for sparse matrix multiplication on the gpu," 2018.

[118] F. Seide and A. Agarwal, "CNTK: Microsoft's Open-Source Deep-Learning Toolkit." ACM, pp. 2135–2135.

[119] S. Eliuk, C. Upright, and A. Skjellum, "dmath: A scalable linear algebra and math library for heterogeneous GP-GPU architectures," vol. abs/1604.01416, 2016. [Online]. Available: http://arxiv.org/abs/1604.01416

[120] A. A. Awan, K. Hamidouche, J. M. Hashmi, and D. K. Panda, "S-caffe: Co-designing mpi runtimes and caffe for scalable deep learning on modern gpu clusters," in *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '17. New York, NY, USA: ACM, 2017, pp. 193–205. [Online]. Available: http://doi.acm.org/10.1145/3018743.3018769

[121] S. Mittal and S. Vaishay, "A survey of techniques for optimizing deep learning on gpus," *Journal of Systems Architecture*, vol. 99, p. 101635, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762119302656

[122] C. Li, Y. Yang, M. Feng, S. Chakradhar, and H. Zhou, "Optimizing memory efficiency for deep convolutional neural networks on gpus," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2016, pp. 633–644.

[123] Xiaoming Chen, Jianxu Chen, D. Z. Chen, and Xiaobo Sharon Hu, "Optimizing memory efficiency for convolution kernels on kepler gpus," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2017, pp. 1–6.

[124] X. Li, Y. Liang, S. Yan, L. Jia, and Y. Li, "A coordinated tiling and batching framework for efficient gemm on gpus," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 19. New York, NY, USA: Association for Computing Machinery, 2019, p. 229241. [Online]. Available: https://doi.org/10.1145/3293883.3295734

[125] P. Jain, X. Mo, A. Jain, A. Tumanov, J. E. Gonzalez, and I. Stoica, "The ooo VLIW JIT compiler for GPU inference," *CoRR*, vol. abs/1901.10008, 2019. [Online]. Available: http://arxiv.org/abs/1901.10008

[126] N. Jouppi, "Google supercharges machine learning tasks with tpu custom chip," 2016.

[127] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," 2017.

[128] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha, "A million spiking-neuron integrated circuit with a scalable communication network and interface," vol. 345, no. 6197. American Association for the Advancement of Science, 2014, pp. 668–673.

[129] S. K. Esser, R. Appuswamy, P. Merolla, J. V. Arthur, and D. S. Modha, "Backpropagation for energy-efficient neuromorphic computing," in *NIPS*, 2015, pp. 1117–1125.

[130] S. K. Esser, P. A. Merolla, J. V. Arthur, A. S. Cassidy, R. Appuswamy, A. Andreopoulos, D. J. Berg, J. L. McKinstry, T. Melano, D. R. Barch, C. di Nolfo, P. Datta, A. Amir, B. Taba, M. D. Flickner, and D. S. Modha, "Convolutional networks for fast, energy-efficient neuromorphic computing." National Acad Sciences, 2016, pp. 11 441–11 446.

[131] NVIDIA, "Nvidia tesla v100 gpu architecture," 2017. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[132] Y. Cheng, F. X. Yu, R. S. Feris, S. Kumar, A. Choudhary, and S.-F. Chang, "An exploration of parameter redundancy in deep networks with circulant projections," in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 2857–2865.

[133] C. Deng, S. Liao, Y. Xie, K. K. Parhi, X. Qian, and B. Yuan, "Permdnn: Efficient compressed dnn architecture with permuted diagonal matrices," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2018, pp. 189–202.

[134] J. Turner, J. Cano, V. Radu, E. J. Crowley, M. O'Boyle, and A. Storkey, "Characterising across-stack optimisations for deep convolutional neural networks," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, September 2018, pp. 101–110.

[135] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, June 2017, pp. 548–560.

[136] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 1–13.

[137] Y. Chen, T. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, June 2019.

[138] S. Cao, C. Zhang, Z. Yao, W. Xiao, L. Nie, D. Zhan, Y. Liu, M. Wu, and L. Zhang, "Efficient and effective sparse lstm on fpga with bank-balanced sparsity," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19, 2019, pp. 63–72. [Online]. Available: http://doi.acm.org/10.1145/3289602.3293898

[139] S. Dey, K. Huang, P. A. Beerel, and K. M. Chugg, "Pre-defined sparse neural networks with hardware acceleration," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 332–345, June 2019.

[140] S. Cao, C. Zhang, Z. Yao, W. Xiao, L. Nie, D. Zhan, Y. Liu, M. Wu, and L. Zhang, "Efficient and effective sparse lstm on fpga with bank-balanced sparsity," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: ACM, 2019, pp. 63–72. [Online]. Available: http://doi.acm.org/10.1145/3289602.3293898

[141] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. W. Keckler, "Compressing dma engine: Leveraging activation sparsity for training deep neural networks," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2018, pp. 78–91.

[142] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 58–70.

[143] R. B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M. E. Wazlowski, and P. M. Bland, "Ibm memory expansion technology (mxt)," *IBM Journal of Research and Development*, vol. 45, no. 2, pp. 271–285, 2001.

[144] P. Ramarao, "Cuda 11 features revealed," 2020.

[145] Y. Kwon, Y. Lee, and M. Rhu, "Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 52. New York, NY, USA: Association for Computing Machinery, 2019, p. 740753. [Online]. Available: https://doi.org/10.1145/3352460.3358284

[146] L. Ke, U. Gupta, C.-J. Wu, B. Y. Cho, M. Hempstead, B. Reagen, X. Zhang, D. Brooks, V. Chandra, U. Diril, A. Firoozshahian, K. Hazelwood, B. Jia, H.-H. S. Lee, M. Li, B. Maher, D. Mudigere, M. Naumov, M. Schatz, M. Smelyanskiy, and X. Wang, "Recnmp: Accelerating personalized recommendation with near-memory processing," 2019.

[147] F. Schuiki, M. Schaffner, F. K. Grkaynak, and L. Benini, "A scalable near-memory architecture for training deep neural networks on large in-memory datasets," *IEEE Transactions on Computers*, vol. 68, no. 4, pp. 484–497, 2019.

[148] Google, "How to use gflags," 2016.

[149] ——, "How to use google logging library," 2016.

[150] N. Corporation, "CuDNN library," 2016.

[151] NVIDIA, "Tesla gpu accelerators for servers," 2016.

[152] ——, "NVIDIA's Next Generation CUDA™ Compute Architecture, Kepler™ GK110," 2012.

[153] ——, "Cuda toolkit documentation," 2016.

[154] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with cots hpc systems," *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, vol. 28, pp. 1337–1345, 2013.

[155] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[156] N. Corporation, "CuBlas library v7.5," 2015.

[157] NVIDIA, "NVIDIA GeForce GTX 1080," 2016.

[158] ——, "NVIDIA Tesla P100," 2016.

[159] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," *Performance Analysis of Systems and Software (ISPASS)*, 2010.

[160] Y. Tian, S. Puthoor, J. L. Greathouse, B. M. Beckmann, and D. A. Jiménez, "Adaptive gpu cache bypassing." New York, NY, USA: ACM, 2015.

[161] G. Wang, Y. Lin, and W. Yi, "Kernel fusion: An effective method for better power efficiency on multithreaded gpu." Washington, DC, USA: IEEE Computer Society, 2010.

[162] B. L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *Proceedings of the IEEE*, vol. 108, no. 4, pp. 485–532, 2020.

[163] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *J. Mach. Learn. Res.*, vol. 18, no. 1, p. 68696898, Jan. 2017.

[164] J. Cheng, J. Wu, C. Leng, Y. Wang, and Q. Hu, "Quantized cnn: A unified approach to accelerate and compress convolutional networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 29, no. 10, pp. 4730–4743, 2018.

[165] NVIDIA, "Nvidia dgx a100 the universal system for ai infrastructure," 2020.

[166] AMD, "Radeon Instinct(TM) MI50 Accelerator," 2019.

[167] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," in *Computer Vision – ECCV 2016*. Cham: Springer International Publishing, 2016, pp. 630–645.

[168] T.-H. Yang, H.-Y. Cheng, C.-L. Yang, I.-C. Tseng, H.-W. Hu, H.-S. Chang, and H.-P. Li, "Sparse reram engine: Joint exploration of activation and weight sparsity in compressed neural networks," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: ACM, 2019, pp. 236–249. [Online]. Available: http://doi.acm.org/10.1145/3307650.3322271

[169] L. Cavigelli and L. Benini, "Extended bit-plane compression for convolutional neural network accelerators," *CoRR*, vol. abs/1810.03979, 2018. [Online]. Available: http://arxiv.org/abs/1810.03979

[170] AMD, "rocblas," 2019. [Online]. Available: https://rocmdocs.amd.com/en/latest/ROCm_Tools/rocblas.html

[171] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009.

[172] J. Gilbert, C. Moler, and R. Schreiber, "Sparse matrices in matlab: Design and implementation," *SIAM Journal on Matrix Analysis and Applications*, vol. 13, no. 1, pp. 333–356, 1992.

[173] J. Khan, P. Fultz, A. Tamazov, D. Lowell, C. Liu, M. Melesse, M. Nandhimandalam, K. Nasyrov, I. Perminov, T. Shah, V. Filippov, J. Zhang, J. Zhou, B. Natarajan, and M. Daga, "Miopen: An open source library for deep learning primitives," 2019.

[174] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," in *International Conference on Learning Representations*, 2018.

[175] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL: Revised OpenCL 1.2 Edition*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.

[176] J. H. Ahn, M. Erez, and W. J. Dally, "The design space of data-parallel memory systems," in *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006, pp. 2–2.

[177] AMD, "Amd gcn3 isa architecture manual," 2016. [Online]. Available: https://gpuopen.com/wp-content/uploads/2016/08/AMD_GCN3_Instruction_Set_Architecture_rev1.1.pdf

[178] AMD, "Radeon(TM) RX Vega56 Graphics," 2017. [Online]. Available: https://www.amd.com/en/products/graphics/radeon-rx-vega-56

[179] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009.

[180] Y. Sun, T. Baruah, S. A. Mojumder, S. Dong, X. Gong, S. Treadway, Y. Bao, S. Hance, C. McCardwell, V. Zhao, H. Barclay, A. K. Ziabari, Z. Chen, R. Ubal, J. L. Abellán, J. Kim, A. Joshi, and D. Kaeli, "Mgpusim: Enabling multi-gpu performance modeling and optimization," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: ACM, 2019, pp. 197–209. [Online]. Available: http://doi.acm.org/10.1145/3307650.3322230

[181] AMD, "Amd radeon instinct mi6 accelerator," 2018. [Online]. Available: https://www.amd.com/en/products/professional-graphics/instinct-mi6

[182] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," *ArXiv*, vol. abs/1603.07285, 2016.

[183] AMD, "Amd app sdk 3.0 getting started," 2017.

[184] S. Liao, Z. Li, X. Lin, Q. Qiu, Y. Wang, and B. Yuan, "Energy-efficient, high-performance, highly-compressed deep neural network design using block-circulant matrices," in *Proceedings of the 36th International Conference on Computer-Aided Design*. IEEE Press, 2017, pp. 458–465.

[185] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang, "Accelerating binarized convolutional neural networks with software-programmable fpgas," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 15–24.

[186] H. Alemdar, N. Caldwell, V. Leroy, A. Prost-Boucle, and F. Pétrot, "Ternary neural networks for resource-efficient AI applications," vol. abs/1609.00222, 2016. [Online]. Available: http://arxiv.org/abs/1609.00222

[187] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, p. 17351780, Nov. 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[188] L. Zhao, S. Liao, Y. Wang, Z. Li, J. Tang, and B. Yuan, "Theoretical properties for neural networks with weight matrices of low displacement rank," in *International Conference on Machine Learning*, 2017, pp. 4082–4090.

[189] V. Sindhwani, T. Sainath, and S. Kumar, "Structured transforms for small-footprint deep learning," in *Advances in Neural Information Processing Systems*, 2015, pp. 3088–3096.

[190] Y. Wang, C. Ding, Z. Li, G. Yuan, S. Liao, X. Ma, B. Yuan, X. Qian, J. Tang, Q. Qiu, and X. Lin, "Towards ultra-high performance and energy efficiency of deep learning systems: an algorithm-hardware co-optimization framework," in *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI)*, 2018.

[191] V. Y. Pan, *Structured matrices and polynomials: unified superfast algorithms*. Springer Science & Business Media, 2012.

[192] D. Bini, V. Pan, and W. Eberly, "Polynomial and matrix computations volume 1: Fundamental algorithms," vol. 38, no. 1. Philadelphia, Society for Industrial and Applied Mathematics., 1996, pp. 161–164.

[193] NVIDIA, "Cuda cufft library," 2016.

[194] K. R. Rao, D. N. Kim, and J.-J. Hwang, *Fast Fourier Transform - Algorithms and Applications*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[195] J. Garofolo, L. Lamel, W. Fisher, J. Fiscus, D. Pallett, N. Dahlgren, and V. Zue, "Timit acoustic-phonetic continuous speech corpus," *Linguistic Data Consortium*, 11 1992.

[196] J. Michalek and J. Vanek, "A survey of recent dnn architectures on the timit phone recognition task," 2018.

[197] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010.

[198] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning," in *NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011*, 2011.

[199] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 2672–2680. [Online]. Available: http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf

[200] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009.

[201] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," 2014.

[202] C. Deng, S. Liao, Y. Xie, K. K. Parhi, X. Qian, and B. Yuan, "Permdnn: Efficient compressed dnn architecture with permuted diagonal matrices," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2018, pp. 189–202.