View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by SJSU ScholarWorks

San Jose State University
SJSU ScholarWorks

Master's Theses | Master's Theses and Graduate Research

Spring 2017

Convolutional Neural Network Acceleration on GPU by Exploiting Data Reuse

Sindhuja Gopalakrishnan Elango
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_theses

Recommended Citation

Gopalakrishnan Elango, Sindhuja, "Convolutional Neural Network Acceleration on GPU by Exploiting Data Reuse" (2017). *Master's Theses*. 4800.
DOI: https://doi.org/10.31979/etd.9b4r-na7x
https://scholarworks.sjsu.edu/etd_theses/4800

This Thesis is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Theses by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

CONVOLUTIONAL NEURAL NETWORK ACCELERATION ON GPU BY EXPLOITING DATA REUSE

A Thesis

Presented to

The Faculty of the Department of Computer Engineering

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Sindhuja Gopalakrishnan Elango

May 2017

© 2017

Sindhuja Gopalakrishnan Elango

ALL RIGHTS RESERVED

The Designated Thesis Committee Approves the Thesis Titled

CONVOLUTIONAL NEURAL NETWORK ACCELERATION ON GPU BY EXPLOITING DATA REUSE

by

Sindhuja Gopalakrishnan Elango

APPROVED FOR THE DEPARTMENT OF COMPUTER ENGINEERING

SAN JOSÉ STATE UNIVERSITY

May 2017

Dr. Hyeran Jeon	Department of Computer Engineering
Dr. Younghee Park	Department of Computer Engineering
Dr. Kaikai Liu	Department of Computer Engineering

ABSTRACT

CONVOLUTIONAL NEURAL NETWORK ACCELERATION ON GPU BY EXPLOITING DATA REUSE

by Sindhuja Gopalakrishnan Elango

Graphical processing units (GPUs) achieve high throughput with hundreds of cores for concurrent execution and a large register file for storing the context of thousands of threads. Deep learning algorithms have recently gained popularity for their capability for solving complex problems without programmer intervention. Deep learning algorithms operate with a massive amount of input data that causes high memory access overhead. In the convolutional layer of the deep learning network, there exists a unique pattern of data access and reuse, which is not effectively utilized by the GPU architecture. These abundant redundant memory accesses lead to a significant power and performance overhead. In this thesis, I maintained redundant data in a faster on-chip memory, register file, so that the data that are used by multiple neurons can be directly fetched from the register file without cumbersome system memory accesses. In this method, a neuron's load instruction is replaced by a shuffle instruction if the data are found from the register file. To enable data sharing in the register file, a new register type was used as a destination register of load instructions. By using the unique ID of the new load destination registers, neurons can easily find their data in the register file. By exploiting the underutilized register file space, this method does not impose any area or power overhead on the register file design. The effectiveness of the new idea was evaluated through exhaustive experiments. According to the results, the new idea significantly improved performance and energy efficiency compared to baseline architecture and shared memory version solution.

ACKNOWLEDGMENTS

I would like to express my heartfelt gratitude to Dr. Hyeran Jeon whose expert guidance, enthusiasm, and motivation kept me engaged in my research. She stirred me in the right direction, gave me complete freedom during every phase of the research and never hesitated to help me in times of difficulty.

I would also like to thank Dr. Younghee Park and Dr. Kaikai Liu for their valuable suggestions and for serving on the defense committee.

I genuinely thank my parents, sister, family members and my friends for their love, support, and prayers that kept me moving forward.

I am indebted to my husband Sivaprakash for supporting my dreams, cheering me throughout and playing a major part in my journey.

TABLE OF CONTENTS

List of Tables ix

List of Figures x

1 Introduction 1

2 Background 4

2.1 GPU Architecture 4

2.1.1 GPU Memory Hierarchy 5

2.1.2 GPU Register File Design 6

2.1.3 Shuffle Instructions 8

2.2 Deep Neural Network 8

2.2.1 Convolutional Neural Network 9

2.2.2 Data Usage Pattern of the Convolutional Layer 10

3 Motivation 16

3.1 Significant Data Access Overhead of Deep Learning Applications 16

3.2 On-chip Memory for Redundant Data Storage 17

4 Problem Statement 20

5 Related Work 21

5.1 Energy Efficient GPU Computing 21

5.1.1 Semantics-Aware Cache for Efficient GPU Offloading 21

5.1.2 Compiler Optimizations for Data Movement Overhead 21

5.1.3 Affine Register File Design for Energy Efficiency 22

5.1.4 Optimizing Register Allocation and Thread-Level Parallelism . . . 22
5.1.5 Optimizing Register Utilization Mechanisms 22
5.2 Deep Learning Computing 23
5.2.1 Instruction Set Architecture for Neural Network Applications . . . 23
5.2.2 Blocked Convolutional Computation 24
5.2.3 Spatial Architecture for Efficient Data Flow 24
6 Proposed Idea 25
6.1 NN Benchmark and Dataset 25
6.2 Intra-Warp Data Sharing 26
6.3 Inter-Warp Sharing 29
6.4 L Register, a New Register Type 29
6.5 Corner Cases 30
6.5.1 Intra-Warp Boundary Thread 30
6.5.2 Inter-Warp Boundary Thread 31
6.6 Ignoring the Corner Cases 33
7 Architectural and Compiler Support 34
7.1 L Registers 34
7.2 Lifetime Analysis of L Register 34
7.3 Intra-Warp Sharing 35
7.3.1 Horizontal Sharing 35
7.3.2 Vertical Sharing 36
7.4 New Active Mask for Overlapped Region 37
7.5 Handling Corner Cases 37

7.5.1 Intra-Warp Boundary 37
7.5.2 Inter-Warp Boundary 38
8 Experimental Results 40
8.1 GPGPU-Sim Setup and Modification 40
8.2 Ignoring the Corner Case Threads 40
8.3 Performance 41
8.4 Load Instruction Reduction 44
8.5 Power Dissipation 46
8.6 On-Chip Memory Latency Sensitivity 47
8.7 Scheduler Sensitivity 48
9 Future Work 50
10 Conclusion 51
References 52

List of Tables

Table 1 Comparison of Different Types of GPU Memory 5

List of Figures

Figure 1 Seven-layer convolutional neural network 10

Figure 2 Fully connected layer (left) and convolutional layer (right) 11

Figure 3 Receptive field of first neuron 12

Figure 4 Receptive field of second neuron 13

Figure 5 Input data sharing between two neurons 14

Figure 6 Kernel execution breakdown of TensorFlow's Cifar-10 application . 17

Figure 7 Shared memory size per thread block 19

Figure 8 Horizontal overlapping of input data in adjacent threads 27

Figure 9 Reuse of data using shuffle instruction in horizontal sharing 27

Figure 10 Vertical overlapping of input data in adjacent threads 28

Figure 11 Reuse of data using shuffle instruction in vertical sharing 28

Figure 12 Intersection of horizontal and vertical overlapping 28

Figure 13 Inter-warp sharing of data between threads of different warps 29

Figure 14 Intra-warp corner case of boundary threads 31

Figure 15 Horizontal inter-warp corner case of last thread in a warp 31

Figure 16 Vertical inter-warp corner case of threads 32

Figure 17 Representation of all corner cases in the thread block 32

Figure 18 Reduced image dimension due to skipping of corner cases 33

Figure 19 Destination L register identifiers for input data and weights 35

Figure 20 Convolutional kernel dimension and stride 36

Figure 21 Solution for intra-warp boundary corner case thread 38

Figure 22 Solution for horizontal inter-warp corner case of boundary thread . . 39

Figure 23 Mean square error of each layer after skipping corner cases 41

Figure 24 Comparison of performance of shared memory and L register without corner case 41

Figure 25 Comparison of performance of shared memory and L register with corner case 42

Figure 26 Comparison of performance of shared memory and L register without corner case using three clusters 43

Figure 27 Comparison of performance of shared memory and L register with corner case for large image dimension of 33*33 43

Figure 28 Comparison of performance of shared memory and L register with corner case for Cifar-10 input image 44

Figure 29 Comparison of load instruction count of shared memory and L register without corner case over baseline 44

Figure 30 Comparison of load instruction count of shared memory and L register with corner case over baseline 45

Figure 31 Comparison of load instruction count of shared memory and L register with corner case over baseline for large image dimension of 33*33 45

Figure 32 Comparison of load instruction count of shared memory and L register with corner case over baseline for Cifar-10 input image . . . 46

Figure 33 Comparison of average power dissipation of baseline and L register without corner case 47

Figure 34 Comparison of latency sensitivity of baseline, shared memory, L register with corner case and L register without corner case 48

Figure 35 Scheduler sensitivity of L register for TLA and GTO warp schedulers normalized over LRR scheduler 49

CHAPTER 1

Introduction

Deep learning applications offer solutions to a wide spectrum of real-world problems. As an artificially intelligent system, deep learning has the potential to interpret sensory data in a human-like manner and deliver useful results. Recently, deep learning has gained great popularity due to its potential to achieve exceptional accuracy in various computer vision applications [1]. Deep learning networks deal with multi-dimensional input data and perform highly complex linear algebraic operations on the data [2]. The networks are extremely convoluted and pose challenges to the underlying processors in achieving throughput and satisfying memory requirements. In traditional processors such as the central processing units (CPUs), the number of processing units is limited [3]. Therefore, CPUs are inefficient for deep learning applications due to lack of parallel processing power. Graphics processing units (GPUs), on the other hand, are inherently parallel with hundreds of cores to process data and high bandwidth memories to store data and results [4]. The massive arithmetic throughput and memory bandwidth of the GPU make it an ideal choice for deep learning applications that exploit data parallelism [7, 8, 10]. The convolutional neural network (CNN), a type of deep neural network (DNN), has convolutional layers in which neurons in the layer calculate their output using a window of input data [8]. When a neuron calculates its output, it reuses a part of the adjacent neuron's input data. In addition, the weights assigned to the inputs of all neurons are the same throughout the layer. Hence input data and weights are often reused and shared across neurons [19]. However, the current GPU architecture does not support this sharing. In GPUs, hundreds or even thousands of neurons are processed concurrently by employing that many hardware threads, where individual threads deal with a neuron's work. Therefore, individual neurons fetch their input and weight values separately without exploiting the data redundancy. This redundant data access lead to excessive memory usage and energy consumption.

The main focus of this thesis was to improve the performance of the convolutional layer in deep neural networks by focusing on minimal input data movement without sacrificing accuracy. Since convolutional layers are the computationally intensive layers and heart of CNNs [11], the performance enhancement in these layers brought a positive impact on the neural network as a whole. The contributions of this work were as follows.

1. To the best of my knowledge, this is the first study that exploits data sharing across neurons in the micro-architecture level of GPUs. Instead of redundant memory accesses to the slow system memories, I exploited underutilized register file space to maintain data that could be shared. By using a simple register mapping algorithm, neurons can fetch data from the register file if the data have been already fetched by another neuron.
2. A new register type, namely L register, was introduced for storing the input data of a thread. With the new registers, it is possible to allow data reuse across neurons at the register level by retaining the sharable data in the registers until its usage is completely exhausted or terminated. The shuffling logic introduced by Kepler enables the neurons to share L register values of adjacent neurons. With this approach, redundant memory accesses as well as load-store unit computations were minimized.
3. The L register's lifetime was analyzed and each register was released once the thread itself and its neighbors had completed the last access to the register. This step is critical because the newly proposed L registers should not induce any additional register pressure. To release the dead registers, the access patterns of individual threads were monitored dynamically, and the last read to the register was determined to declare the register as dead.
4. The performance of the L register design was compared with the shared memory version. The L register design outperformed the shared memory version when larger images and more feature maps were used. In GPUs, the size of shared memory is smaller than the

register file. Therefore, the shared memory version could not scale well with larger input data.

The remainder of the thesis is organized as follows. Chapters 2 and 3 describe the background and motivational data. Chapter 4 gives an overview of the problem statement. Chapter 5 covers the prominent studies that were done previously for energy efficient GPU computing and deep learning computing. Chapter 6 elaborates the core idea of this thesis. Chapter 7 describes the implementation details. Chapter 8 gives the experimental results. Finally, chapter 9 and 10 discuss the conclusion and the future work.

CHAPTER 2

Background

2.1 GPU Architecture

GPU is a parallel computing device and functions as a co-processor for the CPU [12]. The CPU offloads compute heavy portions of an application to the GPU. The CPU is the host that invokes the kernel functions on the GPU which is the device. Data transfer between the host and the device is through the PCI-E bus. GPUs consist of streaming multi-processors (SMs) with each SM having 32 single instruction multiple thread (SIMT) lanes [13]. Each SIMT lane is executed on a compute unified device architecture (CUDA) core. For example, an NVIDIA Fermi architecture consists of 16 SMs, each containing 32 CUDA cores [4]. In total, there are 512 CUDA cores. Each core has an arithmetic and logical unit (ALU) that executes arithmetic and logical operations and a floating point unit (FPU) that executes single- and double-precision floating point operations. Each SM also has four special function units (SFUs) for carrying out transcendental operations and 16 load-store units (LDs) for calculating memory addresses.

The kernel functions invoked by the CPU consist of independent cooperative thread arrays (CTAs) or thread blocks [14]. When the program counter (PC) points to an instruction, a group of 32 threads, called warp in NVIDIA terminology, executes that instruction on 32 execution lanes, synchronously accessing different data operands. Scheduler units schedule the warp to the SM in a round-robin fashion. The operands of all the threads in a warp are fetched from a large register file. The register file has a banked structure. Each bank delivers the data for one thread. The data retrieved from the register file are collected in the operand collector unit, and only when data are available for all 32 threads is the warp instruction ready for execution. When there are bank conflicts, an arbiter unit serializes the accesses to the banks and the operand collector unit buffers the data until all threads have collected their data.

2.1.1 GPU Memory Hierarchy

The CPU and the GPU have their own memory, namely, host and device memory. The CPU can read and write to the host and device memory. From the perspective of a CUDA programming model [15], device memory offers six types of memory space: register file, shared memory, local memory, texture memory, constant memory, and global memory. These memory types differ in size, speed, scope, lifetime, accessibility, and location. A brief comparison of the memory types is listed in Table 1.

Table 1: Comparison of Different Types of GPU Memory

Memory Type	Size	Location	Latency
Register Files	128 KB	On-chip	< 5 cycles
Shared Memory	16 KB/SM	On-chip	< 5 cycles
Constant Memory	64 KB	Off-chip	1-100 cycles
Texture Memory	64 KB	Off-chip	1-100 cycles
Global Memory	10 GB	Off-chip	400-800 cycles

The register file is the largest on-chip memory and is shared across the warps running on the same SM. Each thread in a warp has its own set of registers. Shared memory is an on-chip memory and is shared by all threads within the same thread block. Therefore, threads within a block communicate with each other via shared memory. Constant memory stores data that do not change during the course of kernel execution. A single data value loaded from constant memory can be broadcasted to the threads of a warp. Hence, it is useful when all threads of a warp need the same data. Texture memory is typically useful for graphics applications. When the memory access patterns have spatial locality, texture memory can boost the performance of the application. Global memory is an off-chip memory. It is regarded as a system memory for the GPU. Global memory is large and has very high access latency. Local memory is also an off-chip memory and provides a private storage to each thread. It has the same latency and bandwidth as global memory.

The register file is the fastest; shared memory is slower than the register file but faster than global memory. The throughput can be maximized if data transfer from global memory is reduced and on-chip data usage is maximized. Since the register file and shared memory are on-chip memories, they are indeed the best choice to achieve better performance in data access. However, there are certain differences that should be considered for choosing between registers and shared memory. The scope of a register is limited to a thread, and the number of registers allocated per thread is limited. When there are not enough registers, some register values are spilled into local memory. Shared memory has bank conflict issues when multiple threads access the same bank. Another problem with shared memory occurs when threads have to share a large amount of data at once. The entirety of the data cannot fit in shared memory and must be fetched from global memory. Thus, the choice of memory for any application will be a combination of all or few of the memories based on the needs and the type of application. It is pertinent to utilize memory efficiently to boost the performance of the application.

2.1.2 GPU Register File Design

The GPU has a large, unified, on-chip, banked register file structure. The register file in Fermi [4] is 128 KB with 32768 four-byte registers. The register file is partitioned into 32 banks, with each bank comprising 1024 physical registers. For a single warp instruction, a warp of 32 threads can access the 32 banks independently by using a single read port and a single write port available for each bank. Thus, each bank simultaneously delivers the data for each thread, making the register file a low latency and high bandwidth structure. Generally, the registers are allocated to a warp on a per thread basis. At full occupancy, when there is a maximum of 48 warps, the number of registers per thread is limited to just 21. If an application uses fewer than 48 warps, each thread can use up to 63 registers. The registers are private storage for each thread and cannot be accessed by other threads. When

a thread block has to store a large number of variables and the variables have an extended lifetime, some of the register contents spill to local memory. This is called register spilling. Register spilling decreases the performance of the application, as the variables have to be saved to and restored back from the device's local memory. If the occupancy level is too low, execution time may increase especially when instructions have dependencies. The pipeline latency cannot be hidden without enough number of warps between two consecutively scheduled dependent instructions. To efficiently utilize the register file, a recent study proposed to use register renaming [28]. When register renaming is used, individual architectural registers are dynamically mapped to physical registers. The mapping between the architectural and physical register is created whenever a new value is written to a register. The mapping is released and remapped when a new instance of the architectural register is created.

The register file maintains the context of all active warps during the execution. Thus, warps can retrieve their data from the register file without expensive context switching [4]. To support more active warps, the latest generations of GPUs have been equipped with larger register files [4, 5, 6]. When observing the generation of NVIDIA GPUs starting from the G80 up to the very recent Maxwell, one common trend is the increase in the number of CUDA cores and the size of the register file. Every GPU design aims to increase efficiency and performance of an application by incorporating additional computing units, larger register files, and techniques for performing the toughest computations. Though a large register file is present, some applications are hindered by limited parallelism or a small code footprint that causes the register files to be underutilized. In this thesis, the proposed L registers made use of this underutilized register file to store the input data. Hence, additional on-chip memory structures were not needed for the design.

2.1.3 Shuffle Instructions

CUDA programs have special functions referred to as warp shuffle functions [15] through which threads can communicate with each other. Devices with compute capability of 3.x or higher support these functions. Using these functions, threads can read register values of other threads within a warp. When a shuffle instruction is executed, a 4-byte data is moved from one lane to the other for all the threads in the warp. For example, when __shfl(var, delta, width) is executed by the Nth thread in a warp, the value in the register designated by var is forwarded to the (N+delta)th thread in the same warp.

2.2 Deep Neural Network

Deep neural network (DNN) [8] is one technique of implementing machine learning and has created an explosion in the development of artificial intelligence. In machine learning, a system successfully accomplishes complex tasks with simple algorithms and minimal engineering effort. The structural design of the system replicates the neural network of human brain that comprises neurons and the interface among neurons. A shallow neural network has an input layer, a hidden layer, and an output layer. This is the simplest neural network that was first designed. DNN expands the basic shallow network by configuring it with more hidden layers, making the network deeper and larger. The choice of a shallow or deep network depends on how complex the target problem is. Deep learning is predominantly used in image recognition applications [16, 17, 18] because of it's capability of transforming an input image to better representations across the multiple layers by eliminating irrelevant data at each layer. This learning ability from data without human intervention is the basis of representation learning [8]. Deeper networks are successful in handling huge quantities of data and generating accurate results by learning complex features of the image at their various layers.

In image classification, the deep network has to correctly classify the input image. Instead of using a sophisticated algorithm that is dedicated to a few classes only, the

network gains intelligence through repeated training with a large labeled dataset. This is referred to as supervised learning [9]. Once an input image is fed to the first layer of the network, the features of the image are extracted by the hidden layers that lie between the input and output layer. The output of each layer is an abstract representation of the input image. Finally, the output layer predicts the scores for each category. Known as the forward pass, this is the first phase where the input is propagated in the network to compute the output. The ultimate aim of the deep network is to achieve a very high classification accuracy by minimizing the error between predicted scores and desired scores. This is done by repetitively fine tuning the millions of weights that control the input-output relation of each neuron in all layers while going back up to the first layer. This phenomenon is termed back propagation or backward pass [10]. When the error becomes negligible and the prediction accuracy is satisfactory, the network or model is regarded as trained.

2.2.1 Convolutional Neural Network

Convolutional neural networks (CNNs) gained importance in the early 1990's with several applications developed for image detection, face recognition, character recognition, and handwritten digit recognition based on it. Since then, CNN has been widely used in autonomous robots, self-driving cars, vision systems, and social network applications. CNNs have a series of layers, as in Fig. 1, that take advantage of the raw input data with which they are fed. A brief overview of the functionality of layers [7] is described below.

The first few layers of the CNN are the convolutional and pooling layers. Convolutional layers extract feature maps from the input image in a unit of small sections. In each section of the image, a neuron calculates the weighted average of pixel values in the section. The result is passed to a decision-making activation function that determines the presence or absence of a feature.

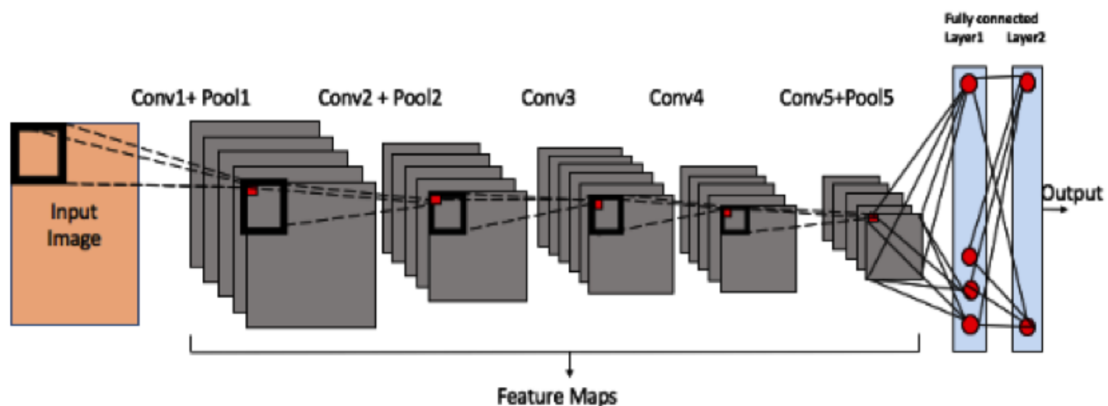

Figure 1: Seven-layer convolutional neural network

Pooling layers merge similar features into one entity, thereby reducing the spatial dimension of the image and making the system invariant to small shifts and disturbances. The last one or two layers are fully connected, which introduces nonlinearity to the outputs of convolutional layers so that input data can be classified into any of the output categories. The difference between the convolutional layers and the fully connected layers is in data connectivity. Fully connected layers connect to all outputs of the previous layer, whereas convolutional layers connect to partial inputs.

2.2.2 Data Usage Pattern of the Convolutional Layer

Of all the layers in CNN, the convolutional layer handles the core functions. It is the most computationally intensive layer and has an architecture that makes it superior for image classification and image recognition tasks. The notable features of the convolutional layer are fourfold:

- Reduced connections
- Local receptive field
- Linearity and shift invariance
- Shared weights and inputs

2.2.2.1 Reduced Connections

In early image classification models, all pixels of the input image were connected to all neurons of the first layer [8]. This fully connected pattern prevailed throughout the successive layers until the final output layer. Fig. 2 shows the connection pattern of fully connected layers and convolutional layers. Imagine an input image to such a fully connected layer as a line of pixels in one dimension. If there are 100 pixels in the input image and the first layer has 150 neurons, then there would be 100 * 150 connections between input image pixels and first layer neurons. If the second layer has 200 neurons, it would result in 150 * 200 connections between the first layer neurons and second layer neurons. Thus, the number of connections increases as we go deeper in the network.

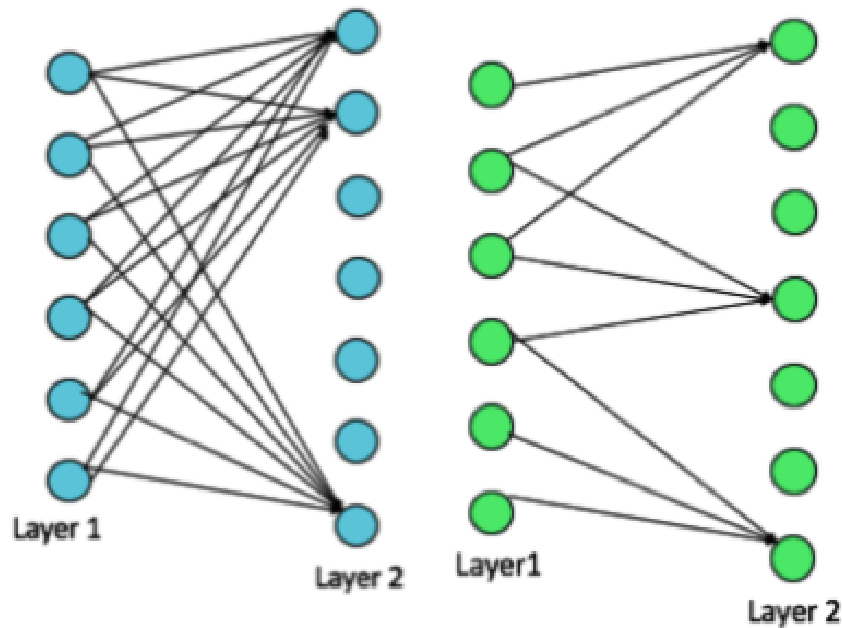

Figure 2: Fully connected layer (left) and convolutional layer (right)

Large numbers of connections significantly reduce the network speed during the training phase. At one point, the training can also become practically impossible because of the huge memory consumption required for storing the network parameters. In addition, a fully connected structure does not consider the spatial arrangement of pixels in the image. It

treats the pixels equally, regardless of the spatial distance between them. Although such a structure performs well, the classification is not based on the spatial structure of images, which is a disadvantage. Convolutional layers rectify these drawbacks [20]. Convolutional layers have a special structure that accelerates the training process with fewer connections, reduces memory consumption, and is well adapted to producing better classification results.

2.2.2.2 Local Receptive Field

Instead of connecting all pixels of the image to a neuron, the convolutional layers restrict the connection to only a region of the image. This local region is the receptive field of the neuron. In Fig. 3, the input image is a two-dimensional image of 6*6 pixels, and the receptive field is a two-dimensional window of size 3*3. The receptive field influences the decision of a neuron. Each pixel in the receptive field has a unique weight value that controls the pixel's contribution to the output. A set of weights is referred to as a weight filter or a kernel, as illustrated in Fig. 3.

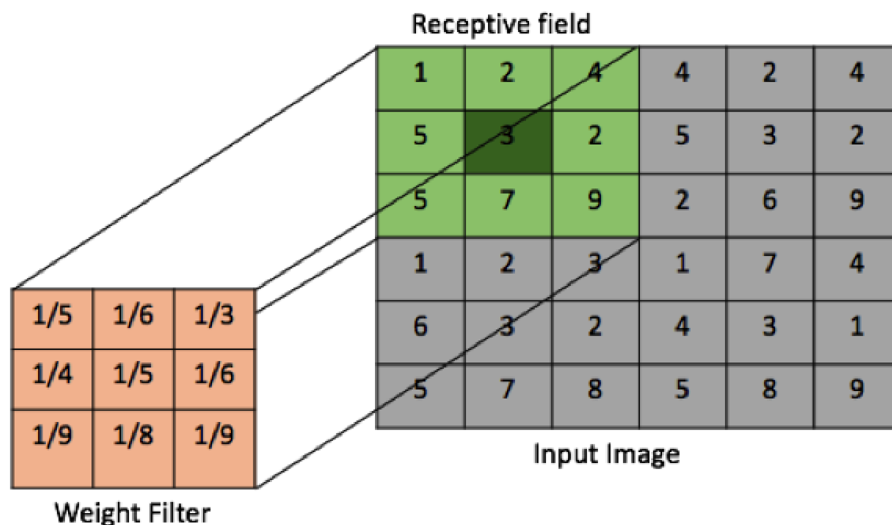

Figure 3: Receptive field of first neuron

An input image is a discrete buffer of pixels. A kernel slides over the buffer and performs operations to extract information from the image. As the kernel slides over each pixel, the receptive field changes accordingly, as indicated in Fig. 4.

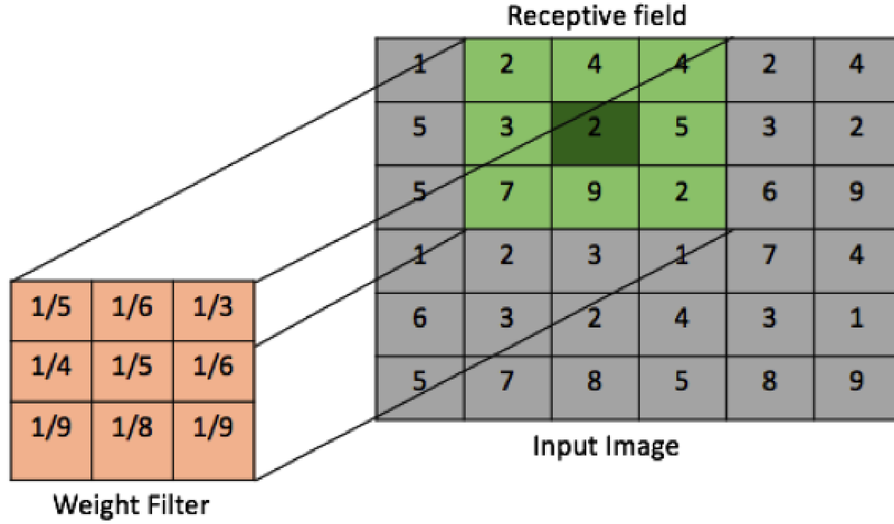

Figure 4: Receptive field of second neuron

2.2.2.3 Linearity and Shift Invariance

Convolutional layer performs a simple local averaging operation on the input image. Two important features of the operation are shift invariance and linearity [19]. Shift invariance means that the same operation repeats on every pixel of the input image. Linearity means that every pixel in the image is replaced with a linear combination of itself and its neighboring pixels. Consider a window of nine pixels whose values are as shown in Fig. 3. In local averaging, a neuron replaces the pixel value 3 at the center of the highlighted receptive field by an average of its immediate surrounding pixels. This involves an addition followed by a division operation.

$$O(5) = (1 + 2 + 4 + 5 + 3 + 2 + 5 + 7 + 9)/9$$

The above step can also be expressed like below.

$$O(5) = 1/9 + 2/9 + 4/9 + 5/9 + 3/9 + 2/9 + 5/9 + 7/9 + 9/9$$

Every pixel is first scaled down by a factor of nine and is summed to derive the output result. The scaling factor (1/9, 1/9, 1/9, 1/9, 1/9, 1/9, 1/9, 1/9, 1/9) is the weight filter that is multiplied by the image pixels to calculate the output pixel value. The filter can also be distributed as (1/5, 1/6, 1/3, 1/4, 1/5, 1/6, 1/9, 1/8, 1/9) in which each pixel can scale differently. To calculate the new value for pixel 2, the weight filter slides on the image, as illustrated in Fig. 4 and the same operation repeats. This continues until new values replace all the pixels of the input image.

2.2.2.4 Shared Weights and Inputs

A convolutional layer may compute several feature maps. Different feature maps use different weight filters to extract different features. Within a feature map, a feature can appear anywhere in the image. Hence, the same weight filter is shared throughout the feature map. Weight sharing also reduces the number of free parameters in the model and expedites the convergence mechanism [10]. In addition to common weights, there is an overlapping region of common inputs between adjacent neurons calculations, as can be seen in Fig. 5.

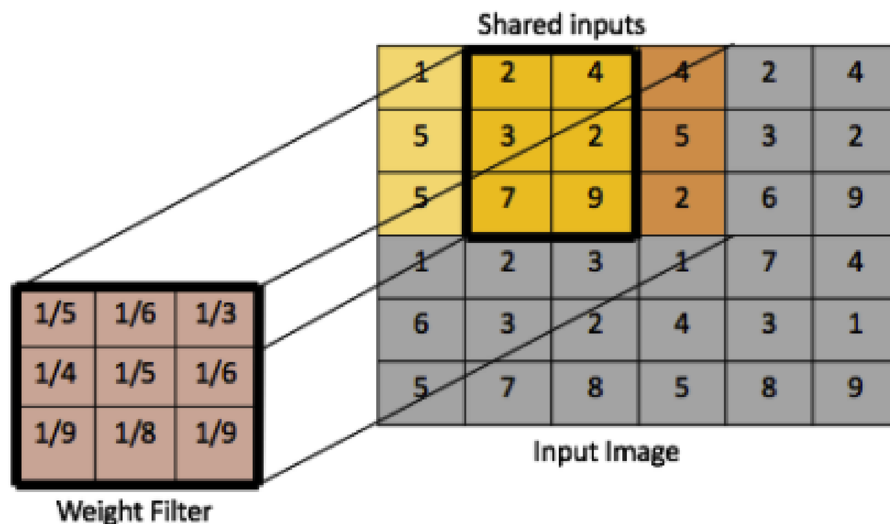

Figure 5: Input data sharing between two neurons

The size of the overlapping region depends on the stride of the weight filter across the image. A stride unit of 1 means there is maximum overlap. As the stride value increases, the degree of overlap decreases. Lower overlap values greatly reduce the spatial dimension of the image. Hence, the choice of stride value depends on the extent of information required by the application.

CHAPTER 3

Motivation

3.1 Significant Data Access Overhead of Deep Learning Applications

In order to identify the performance bottleneck of DNN applications on GPUs, the architectural behavior of popular DNN networks was examined. The evaluation was carried out for four deep learning models, namely LeNet, AlexNet, ImageNet, and RNN LSTM, from three popular deep learning frameworks such as TensorFlow [21], Caffe [23], and Torch [24]. The LeNet model was used for training and testing of handwritten digit recognition. Cifar-10 and AlexNet were used for tiny image classification and LSTM for next character prediction. Three applications were CNN with different depths, inputs, and configurations. All of the applications were profiled using the command line profiler, Nvprof [25], while running on the Tesla Kepler GK210 and MaxWell Tegra X1 GPU architectures. During the total execution time of each application, the frameworks invoked several kernel functions. In the first experiment, the execution time breakdown for each kernel was measured and sorted. The kernel with the longest execution time was picked for analysis.

Two of the three frameworks spent most of the time transferring data from the GPU to the CPU. TensorFlow spent about 60% of the total execution time for data transfer, as depicted in Fig. 6. Abadi et al. [21] explained that this phenomenon of frequent data transfer from the device to the host is done for optimization purposes. Since the size of the global memory is limited, the intermediate values in each layer and the final outputs of each layer are transferred back to the CPU so that the computation size is not limited by GPU memory. Although this is helpful, the intermediate values and results are recomputed by functions in order to be used again by the next layers. This computation seemed inessential and required an optimized solution.

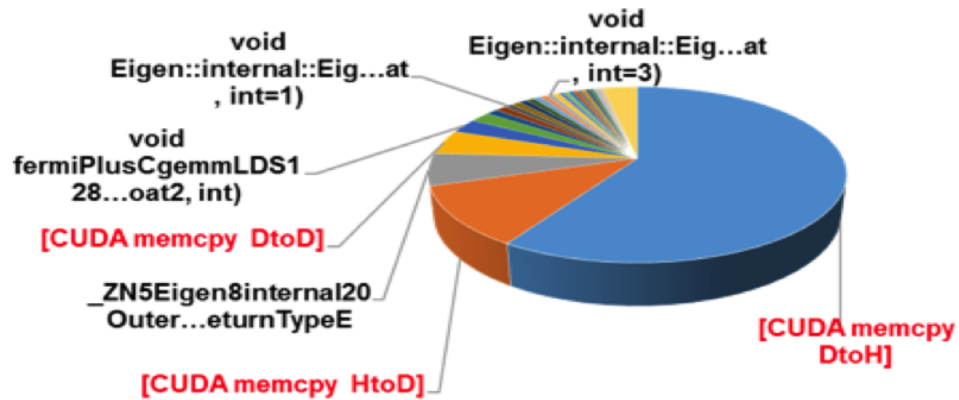

Figure 6: Kernel execution breakdown of TensorFlow's Cifar-10 application

In the next experiment, the batch size of the input given to the neural network varied from 32 to 256, and the memory usage during the execution was measured using the nvidia-smi tool [22]. Note that batch size is the number of inputs given for training or testing concurrently in a neural network. Measuring memory usage is a way to understand the application's data transfer behavior and to distinguish the three frameworks based on their commonalities or differences. While Torch and Caffe's memory usage increased linearly in proportion with an increase in batch size, Tensorflow exhibited a high constant value of memory usage at about 11,000 MB for all batch sizes. This behavior affirmed the conclusion derived from the first experiment, that irrespective of the batch size, data transfer happened frequently in TensorFlow, which led to high memory consumption. Since data usage patterns of convolutional layers facilitate sharing, the redundant data transfer could be minimized if the on-chip memory of the GPU were better utilized.

3.2 On-chip Memory for Redundant Data Storage

In GPUs, there are faster on-chip memories such as a register file, shared memory, and L1 cache. If these on-chip memories can maintain the large input data of DNN applications, system memory may not need to be repeatedly accessed. Several studies have demonstrated that the register file is a large on-chip entity that can improve the performance of the

application if it is optimally exploited [26, 27]. However, not many applications tend to use this resource efficiently because of the limited parallelism that exists in the application's code. In several studies, the underutilization of the register file was observed and exploited the underutilization to improve energy efficiency. Jeon et al. [28] proposed a register management mechanism to efficiently utilize the register file by characterizing the lifetime of registers. Lee et al. [29] examined the redundant data storage in the register file by threads in the same warp and proposed a warped compression and decompression mechanism. The works above focused on the register file access patterns of small benchmark applications. But CNNs typically have a unique data access pattern that requires a customized solution for using the on-chip storage. The new scheme that is presented in the coming sections makes use of the data access pattern of CNNs and the underutilized register files.

One may claim that shared memory can be used for reducing redundant memory accesses. However, shared memory has limited space and is therefore restricted in its ability to hold input data for the applications that use large images and larger batches. For instance, the total shared memory available per SM is 16 KB in Fermi [4]. Shared memory is divided equally among the thread blocks assigned to the SM. The threads of a given thread block can share only the partition of shared memory that is assigned to the thread block. Fig. 7 shows the exponential decrease in shared memory size as the number of thread blocks assigned per SM increases. Many benchmarks and applications actually work on images that have larger dimensions such as 64*64 or 128*128. If the number of thread blocks assigned to the SM increases, then these images cannot be entirely stored in shared memory. This causes a performance degradation, as part of the image has to be fetched again from global memory to shared memory, leading to more load instructions and an increase in the shared memory load-store count. In NN applications, every thread block computes a feature map from the entire image. Therefore, the input image resides in the shared memory space

of every thread block. If the number of thread blocks is high, then multiple copies of the image are fetched to shared memory, which is redundant. Hence, shared memory is not beneficial when the number of threads blocks for an SM is more or when the image size is larger than the available shared memory space per thread block.

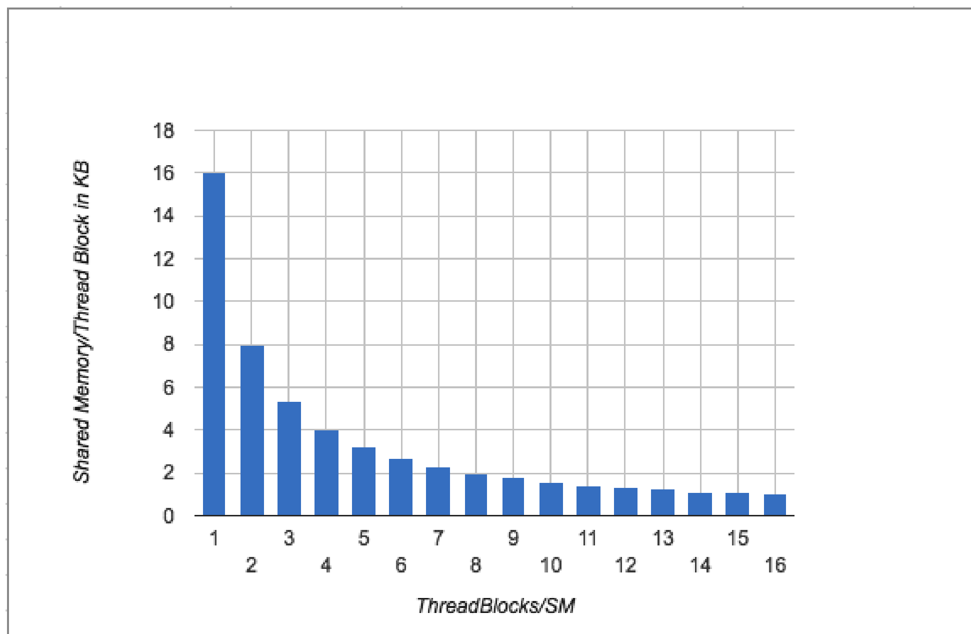

Figure 7: Shared memory size per thread block

CHAPTER 4

Problem Statement

Deep learning applications have much data that could be shared between threads. However, instead of reusing the data, applications fetch data from global memory, which expends much time and energy. To prevent this, data should be available for reuse in the fast on-chip memory. Shared memory is beneficial as it is on-chip and allows threads in thread blocks to communicate. However, shared memory has limited capacity and does not guarantee the best performance when the images used by the application are large.

The register file is a useful, large on-chip resource whose scope is limited to a thread. The access latency of a register file is almost negligible, but the register files are not fully utilized by applications. Hence, the data to be reused can be stored in the register files. However, the existing register file design does not provide the best layout for making use of the data sharing that prevails in deep learning applications.

The recently released instruction, shuffle, makes it possible for threads within the same warp to exchange data between their registers. However, the question then arises as to which registers must retain the data as not all data are necessary.

The next issue is how long the registers should hold the data. If registers hold data for longer than necessary, then a demand for registers is created because each thread has a limited number of registers. Register demand causes content from the register to be spilled to local memory and then brought back from local memory when needed.

The focal point of this research was to address the aforementioned issues by proposing a distinct register file design and supporting mechanisms.

CHAPTER 5

Related Work

5.1 Energy Efficient GPU Computing

5.1.1 Semantics-Aware Cache for Efficient GPU Offloading

This work addresses the huge data transfer overhead in collaborative CPU-GPU systems. Alsaber and Kulkarni [30] have proposed an energy-efficient technique to dynamically manage data offloading to the GPU instead of relying on low-level libraries that track all memory accesses at very fine granularity and make only static decisions. They introduced a software cache, SemCache, that is aware of the semantics of the data and does dynamic offloading to the GPU in variable granularity. The cache design is based on the MSI (modified/shared/invalid) cache coherence protocol. The CPU maintains the state of the cache, and hence transfers from the CPU to GPU are made only when the state of the records in the cache are invalid. Results of expensive functional computation and matrices are stored in the cache after the first call so that repeated calls fetch the results directly from the cache. Thus, redundant communication is avoided, and performance is significantly improved.

5.1.2 Compiler Optimizations for Data Movement Overhead

Wu et al. [31] applied scientific computing techniques such as kernel fusion and kernel fission to reduce data transfer at memory hierarchy and between the CPU and GPU. In kernel fusion, dependent kernels are combined to form a new kernel that avoids intermediate result generation. GPU memory becomes more available, which results in multiple benefits such as more data storage at GPU, reduced PCI-E traffic, and increased temporal data locality. Global memory access also decreases as intermediate data resides in GPU registers. In the kernel fission technique, CUDA kernels are split into multiple segments, with each segment acting as a CUDA stream that executes either concurrently or in a pipelined fashion. A unique CTA handles each segment. When one CTA addresses

computation, other CTAs handle data transfer at the same time. Both kernel fusion and fission address bandwidth limitations of the PCI-E bus.

5.1.3 Affine Register File Design for Energy Efficiency

Wang et al. [32] proposed a new affine register file design that eliminates redundant memory access and computations for scalar executions. The compiler identifies affine instructions and adds tags to them. Two type of registers, the base and stride register, store the affine vector values. Affine ALUs are used for computing the affine instructions. Vector operations are converted to scalar operations and executed on an affine ALU. The computation results are then broadcasted. By this method, the number of register reads, register writes, and ALU operations are greatly reduced.

5.1.4 Optimizing Register Allocation and Thread-Level Parallelism

Xie et al. [33] introduced a coordinated framework CRAT that optimizes register allocation and parallel thread execution. The framework emphasizes a register allocation scheme that allocates registers based on register lifetime analysis and invokes graph-coloring algorithms to allocate variables to registers. Since more registers per thread reduce thread-level parallelism (TLP), the idea focuses on managing the register files that directly affect the TLP. Being aware of the limit of registers per thread, the scheme handles register spills by spilling to the less used on-chip shared memory instead of the off-chip local memory. Effective register allocations thereby help in maintaining optimum thread-level parallelism without causing cache contention, leading to improved performance.

5.1.5 Optimizing Register Utilization Mechanisms

Jeon et al. [28] discovered that even after the registers are allocated, the fraction of the time the registers remain active is significantly short. In the virtualized register file scheme that they proposed, lifetime analysis of the register file is done. A small physical register

file is mapped to a large architected register file space by forcefully releasing the dead registers of a warp and remapping the registers to another warp. Thus, physical register space is shared across warps and demand for the registers is reduced. Their approach of under-provisioning the physical register space also reduced static power consumption. Lee et al. [29] observed that threads of the same warp used data values that were highly similar. This fact led them to propose a warped register compression scheme which allows a base register to store the complete value of a thread and to have the other threads store only the delta difference from the base value in the remaining registers of the bank. During computation, the values are decompressed and used. By this compression and decompression scheme, only fewer register banks are activated, and static and dynamic power are reduced. The works above focus on the problem of improved utilization of register files for many linear algebraic applications and benchmark suites.

5.2 Deep Learning Computing

5.2.1 Instruction Set Architecture for Neural Network Applications

Liu et al. [34] demonstrated that for certain neural network applications, computing resources such as CPU, GPU, and accelerators invest excessive resources for different workloads. Based on a comprehensive analysis, they present a new instruction set architecture (ISA), Cambricon, that provides higher code density than the existing x86, MIPS, and GPU. In Cambricon, complex instructions decompose to low-level operations and high-level functionalities are built from these operations. Instead of a vector register file, data are stored in an on-chip scratchpad memory to support multi-bank access simultaneously without the need for a multi-port structure. Their design focused on data-level parallelism, customized instructions, and on-chip memory usage. Because of the multifaceted approach, the evaluation was successful for a wide range of neural network applications.

5.2.2 Blocked Convolutional Computation

Yang et al. [35] came up with an optimized solution for improving energy efficiency in convolutional layers. Since convolutional layers repeatedly perform the same computation, the order of computation does not matter. Hence the loops in convolution are split into smaller loops and the order of execution of split loops is interchanged. This phenomenon is termed blocking. Whenever a new inner loop is added within a large loop, the shared coefficients for that loop's computation are stored in a buffer. As small memories serve the needed data, the number of data transfers from larger memories to small memories is kept minimum. By doing so, the energy cost associated with the memory access is reduced.

5.2.3 Spatial Architecture for Efficient Data Flow

With a motivation to maximize the data reuse in convolutional and fully connected layers, Chen et al. [36] proposed a new data flow pattern, namely row stationary and a spatial architecture optimized for the data flow. The input data, weights, and the partial sums are stored at their best possible level in the memory hierarchy and reused from that level, taking into account the total energy cost associated for the initial fetch, store, and reuse operations on the data. The high-dimensional convolution primitives are broken into 1-dimensional convolution primitives. Then the primitives are mapped to processing engines (PEs). After the computation, by a sliding window approach another set of primitives that require the same weights and inputs as the previous primitives are mapped to the same PEs. By doing this, excessive data movement is prevented.

CHAPTER 6

Proposed Idea

6.1 NN Benchmark and Dataset

The application that was used for this research is a simple, four-layer, feed-forward CNN benchmark suite that was developed and used in the ISPASS 2009 paper on GPGPU-Sim [37]. It is implemented in the CUDA programming language. The first two layers are convolutional layers, while the last two layers are the fully connected layers. The input image is fed to the first convolutional layer. The first layer extracts six feature maps from the image and feeds them to the second layer. The second convolutional layer computes 50 feature maps from the previous layer's 6 feature maps. The third layer is a fully connected layer which has 100 neurons. All of the neurons in this layer are connected to all neurons in the previous layer. The fourth layer is also a fully connected layer and is the final output layer. It consists of 10 neurons and connects to all 100 neurons in the third layer. Since there are only 10 possible values of digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9), the number of neurons in the final layer is based on the number of classes or categories an input image can fall into. The output layer calculates the probability associated with each category. The class corresponding to the highest probability is the predicted class which may or may not match to the actual class depending on the accuracy of the model. The dataset used by the NN application is the famous MNIST database [38], which has a training set of 60000 examples and testing set of 10000 examples of handwritten digits, both clear and with distortions.

Variations of the application and dataset were considered in order to bring about a comprehensive evaluation. The application was modified to include more layers to evaluate the behavior of deeper networks. The default size of the input image is 29*29. Bigger images of dimension 33*33 and 73*73 of handwritten digits were created for testing the application. A different data set such as Cifar-10, consisting of labels of 10 classes of tiny

images, was given to the application. The RGB values in the Cifar-10 were converted to gray scale. Since batch size is an important factor that influences memory usage, the images were given in batches of 10 and 28 to test the implementation. The shared memory version of the NN application for various images sizes and batch sizes was also implemented.

6.2 Intra-Warp Data Sharing

Generally, the threads in a warp function in a lockstep fashion for the same instruction but for different operands [39]. Each thread works on its specific window of input data in parallel and stores the fetched data in the thread-specific registers. At the time when thread 0 fetches its first input data, thread 1 also fetches its first input data. However, the first data for thread 1 is also the Nth data for thread 0 where N is the sharing distance. Thus, there are shareable data between threads, which can be directly fetched from the register file by using warp shuffle instructions instead of load instructions. Data sharing can be done in both row-wise and column-wise. The row-wise shareable region is the horizontal overlapping region, while the column-wise shareable region is the vertical overlapping region.

In horizontal overlapping, as shown in Fig. 8, the last three columns of thread 0 are overlapped with the regions of thread 1 and thread 2. Column 3 and column 4 of thread 0 are logically the same as column 1 and column 2 of thread 1. Column 5 of thread 0 is the same as column 1 of thread 3. In terms of registers, the data that register 3 of thread 0 needs is actually the data that resides in the register indicated by 1' of thread 1. Similarly, the data for register 5 of thread 0 already resides in the register indicated by 1'' of thread 2. Hence, warp shuffling can be used, as shown in Fig. 9.

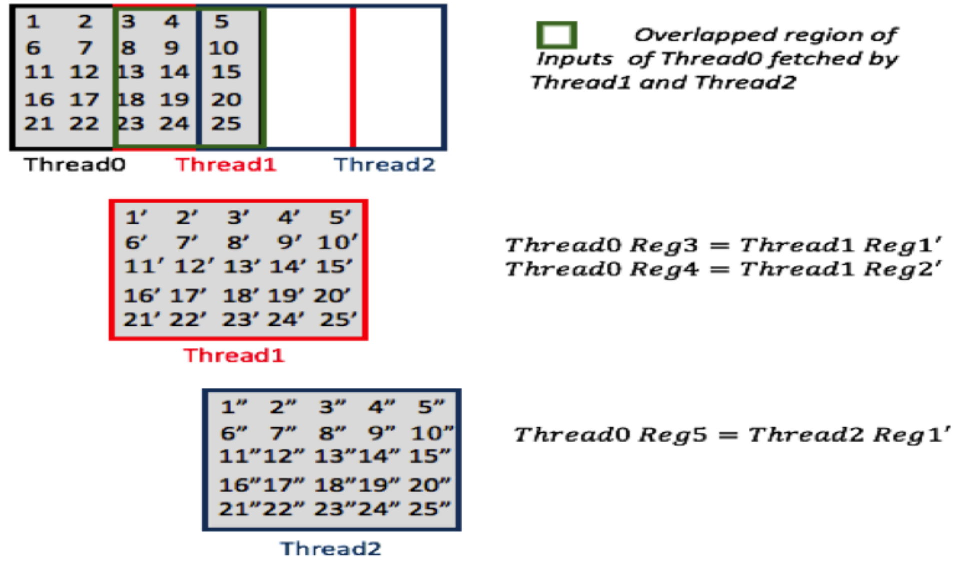

Figure 8: Horizontal overlapping of input data in adjacent threads

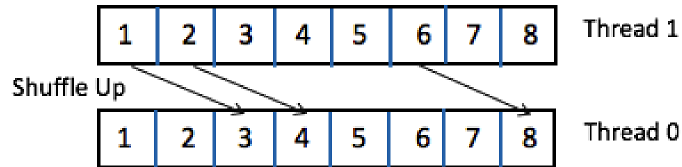

Figure 9: Reuse of data using shuffle instruction in horizontal sharing

In vertical overlapping, as depicted in Fig. 10, the overlapping occurs between the same column of threads in a thread block. In the overlapped region of a single thread, the rows are shared. Row 3 and row 4 of thread 0 are logically the same as row 1 and row 2 of thread 13. Row 5 of thread 0 is the same as row 1 of thread 26. In terms of registers, the data that register 11 of thread 0 needs is actually the data that resides in the register indicated by 1' of thread 13. Similarly, the data for register 21 of thread 0 already reside in the register indicated by 1'' of thread 26. Thread numbers 13 and 26 pick data for thread 0. These thread numbers depend on the dimension of the thread block. The shuffle instruction can be used as shown in Fig. 11.

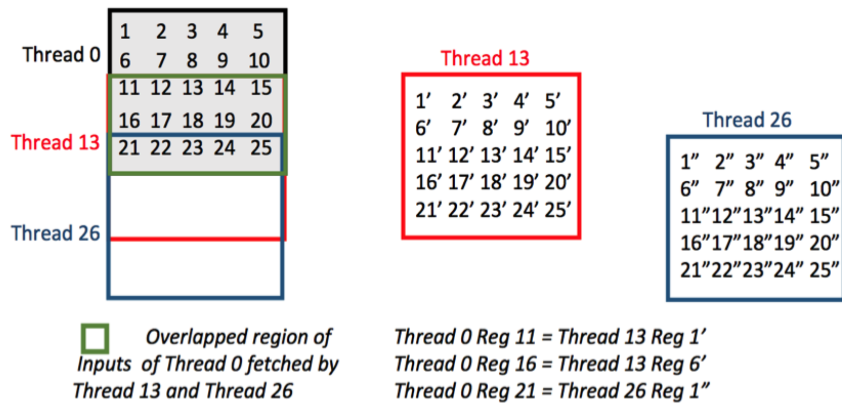

Figure 10: Vertical overlapping of input data in adjacent threads

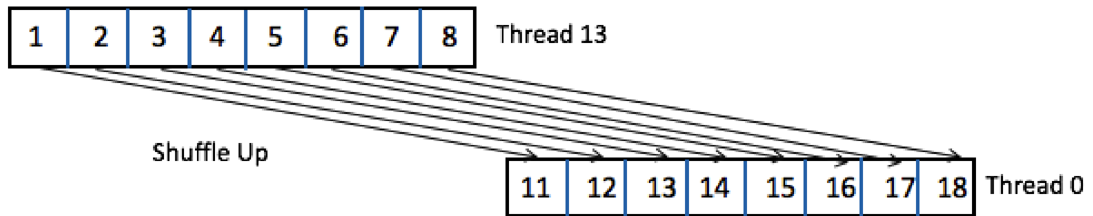

Figure 11: Reuse of data using shuffle instruction in vertical sharing

Thus with horizontal and vertical sharing, every thread can skip many load instructions. The region not overlapping the threads is highlighted with an orange square in Fig. 12. In this region, load instructions cannot be skipped. With this new design, four out of 25 load instructions are executed, which is an 84% reduction in the number of load instructions executed per thread.

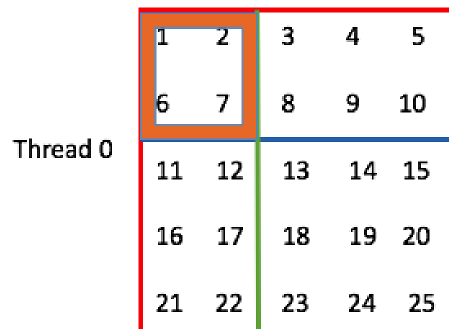

Figure 12: Intersection of horizontal and vertical overlapping

6.3 Inter-Warp Sharing

In Fig. 13, threads in the last row of warp 0 are completely overlapped by threads in the first row of warp 1. This intersection represents the vertical inter-warp sharing that exists between threads of different warps. Unlike threads of the same warp, existing shuffle instructions cannot be used for exchanging contents between threads of different warps. With a new register decoder design proposal, it is possible to perform shuffle across the warps. By using the global thread ID, thread 0 of warp 1 can be considered thread 32 of warp 0 by the register decoder, thereby creating an implied extension for warp 0. However, the research presented here is mainly on intra-warp data sharing. Hence, the inter-warp proposal will be a future expansion of the design.

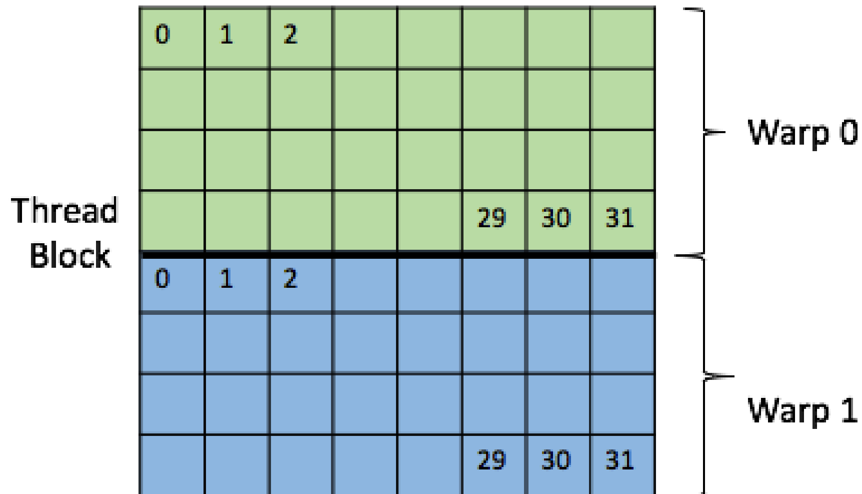

Figure 13: Inter-warp sharing of data between threads of different warps

6.4 L Register, a New Register Type

The load instructions alternatively fetch input pixels and weights from global memory and store them to destination registers. Each load stores the fetched data to destination R type registers such as $R6 for input data and $R8 for weight value. However, the lifetime of the data in the register is very short and ends when the partial sum calculation in each iteration is completed. For subsequent instructions, the same registers store a new data or

result. Destination registers are overwritten and the duration that data reside in them are limited. To prevent this, destination registers for the load instructions should be unique in order to retain the data for later use. Therefore, a new set of uniquely numbered and named special registers are introduced per thread in this design. The load instructions in the generated pseudo-assembly code are identified by using the opcode of the instruction and a special mechanism maps the destination registers to L-type destination registers. This method is applicable only for convolutional layers. A compiler directive indicates the hardware to use L registers at the kernel boundary. When the kernel is launched on the GPU, the compiler directive activates the L register usage. For nonconvolutional layers, the compiler directive is not set. Therefore, L registers are not used here.

6.5 Corner Cases

Horizontal sharing and vertical sharing are applicable for all threads in the thread block except for the corner case threads. The corner case is identified at two levels. The first level is within a warp, classified in the design as the intra-warp boundary thread. The other level is between threads of different warps, classified as the inter-warp boundary thread.

6.5.1 Intra-Warp Boundary Thread

Horizontal sharing is not applicable for the last thread in every row indicated by the red squares in Fig. 14. If the load instructions are skipped for the last thread in every row, then there are no adjacent threads to fetch the input values for this thread. Thus, thread IDs (12, Y), where Y is from 0 to 12, are classified as corner case threads. Similarly, vertical sharing is not applicable for all the threads in the last row of the thread block as indicated by the green squares. Thread IDs (X, 12), where X is from 0 to 12, are also classified as corner case threads. Note that the thread block in this example is two-dimensional and indicated by (X, Y) where the maximum value of X and Y are 12.

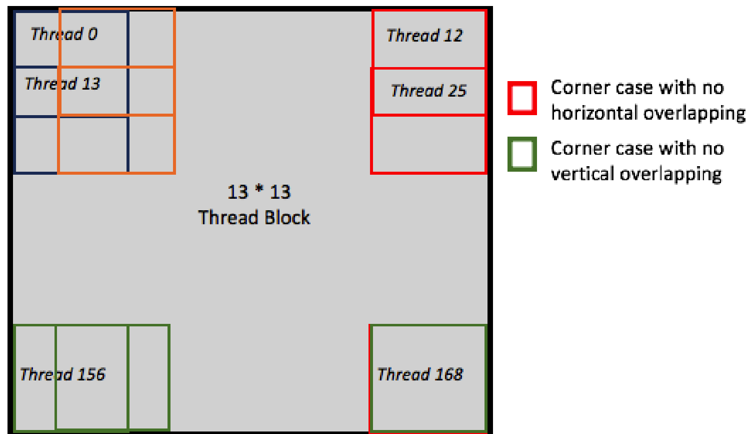

Figure 14: Intra-warp corner case of boundary threads

6.5.2 Inter-Warp Boundary Thread

The total number of threads in the thread block are split into warp units. Each warp has 32 threads. Warps are interleaved and executed by the scheduler. So a thread from warp 1 cannot use the data that is fetched by a thread of warp 2 because the scope of the warp registers is at warp level only unless the hardware is adapted to do it. Hence, load instructions should not be skipped at the inter-warp boundary. Fig. 15 shows horizontal inter-warp boundary where the 31st thread of warp 0 is adjacent to 0th thread of warp 1. For a single thread block, there are n-1 such horizontal inter-warp boundary threads where n is the number of warps in the thread block.

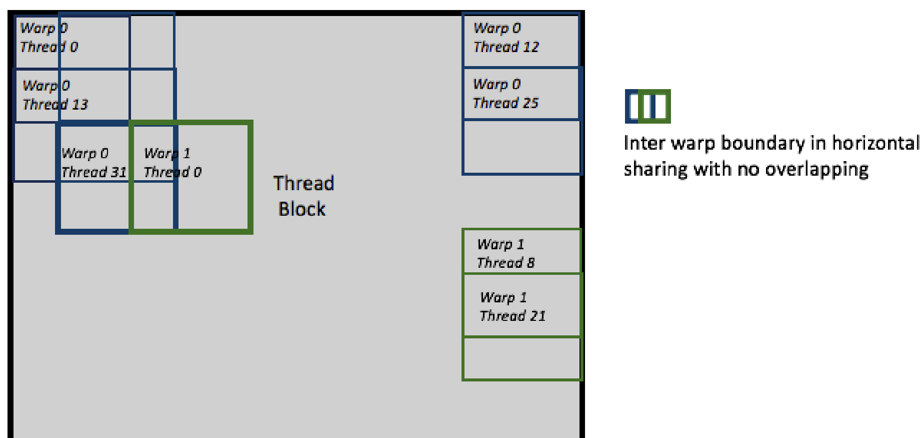

Figure 15: Horizontal inter-warp corner case of last thread in a warp

Fig. 16 shows vertical inter-warp boundary where the threads in almost an entire row of warp 0 are overlapped with threads of warp 1. The number of overlapping threads is even greater than horizontal inter-warp boundary because more threads are intersected. The horizontal and vertical inter-warp boundaries are highlighted represented by the green-blue intersections. Fig. 17 depicts all the corner cases together. Yellow threads represent the intra-warp boundary. Orange and green threads represent the horizontal and vertical inter-warp boundaries, respectively.

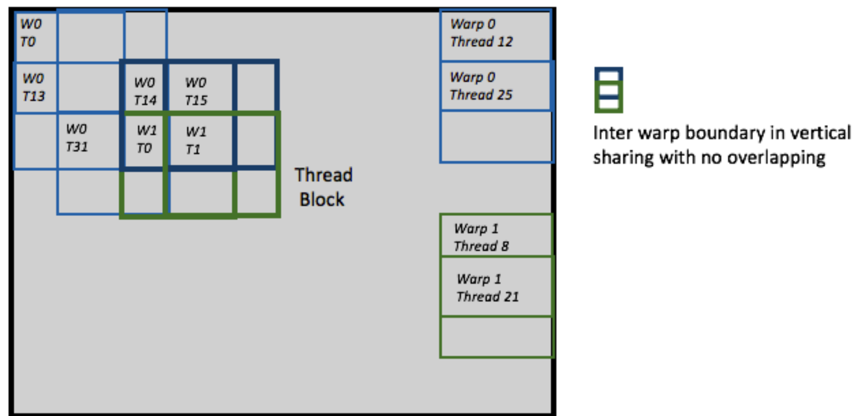

Figure 16: Vertical inter-warp corner case of threads

ThreadIdx.x

ThreadIdx.y

(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)	(6,0)	(7,0)	(8,0)	(9,0)	(10,0)	(11,0)	(12,0)
(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)	(7,1)	(8,1)	(9,1)	(10,1)	(11,1)	(12,1)
(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)	(7,2)	(8,2)	(9,2)	(10,2)	(11,2)	(12,2)
(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)	(7,3)	(8,3)	(9,3)	(10,3)	(11,3)	(12,3)
(0,4)	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)	(7,4)	(8,4)	(9,4)	(10,4)	(11,4)	(12,4)
(0,5)	(1,5)	(2,5)	(3,5)	(4,5)	(5,5)	(6,5)	(7,5)	(8,5)	(9,5)	(10,5)	(11,5)	(12,5)
(0,6)	(1,6)	(2,6)	(3,6)	(4,6)	(5,6)	(6,6)	(7,6)	(8,6)	(9,6)	(10,6)	(11,6)	(12,6)
(0,7)	(1,7)	(2,7)	(3,7)	(4,7)	(5,7)	(6,7)	(7,7)	(8,7)	(9,7)	(10,7)	(11,7)	(12,7)
(0,8)	(1,8)	(2,8)	(3,8)	(4,8)	(5,8)	(6,8)	(7,8)	(8,8)	(9,8)	(10,8)	(11,8)	(12,8)
(0,9)	(1,9)	(2,9)	(3,9)	(4,9)	(5,9)	(6,9)	(7,9)	(8,9)	(9,9)	(10,9)	(11,9)	(12,9)
(0,10)	(1,10)	(2,10)	(3,10)	(4,10)	(5,10)	(6,10)	(7,10)	(8,10)	(9,10)	(10,10)	(11,10)	(12,10)
(0,11)	(1,11)	(2,11)	(3,11)	(4,11)	(5,11)	(6,11)	(7,11)	(8,11)	(9,11)	(10,11)	(11,11)	(12,11)
(0,12)	(1,12)	(2,12)	(3,12)	(4,12)	(5,12)	(6,12)	(7,12)	(8,12)	(9,12)	(10,12)	(11,12)	(12,12)

Figure 17: Representation of all corner cases in the thread block

6.6 Ignoring the Corner Cases

When corner cases are ignored, the inverted L-shaped region of the input image shown in Fig. 18 is not included in the convolution computations. If corner cases are ignored, the threads that belong to this region need to skip 20% of their original computations. The assumption in this approach is that the L region of the image does not have any significant impact on the output, as useful data tends to be in the center of the image. However, this assumption may not be true for all real world applications. Hence, the mean square errors (MSE) of output neurons at each layer of CNN are calculated [40]. MSE is the average value of squares of error or deviation. If the MSE is justifiably low, then the corner cases can be ignored. In the experiment, corner cases of first and second convolutional layers are ignored.

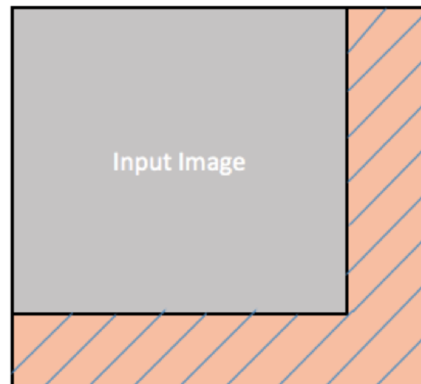

Figure 18: Reduced image dimension due to skipping of corner cases

CHAPTER 7

Architectural and Compiler Support

7.1 L Registers

Since only a few registers of a thread are accessed by other threads, those registers should be uniquely identified. The design introduces a new set of L-type registers that stores only the data fetched by global load instructions. The destination registers used by other instructions of the kernel, like arithmetic and logical instructions, are still the normal R-type registers. L registers significantly reduce design complexity caused by shuffle instructions. It is straightforward to compute the source lane when all input data resides consecutively. If R registers are used instead of the new L registers, then the load destination registers would be interleaved with destination registers of other instructions. Then, it would be almost impossible to identify the destination registers of load instructions that have only the shareable input data. The L register design consists of two phases. First, the compiler uses the L-type designation for the load instructions that fetch input data and weights. Next, regardless of the register ID given by compiler to L registers, the hardware tracks the sequence of L registers for a thread, renames them in increasing order, and stores the weights and input image pixels alternatively.

7.2 Lifetime Analysis of L Register

Like other architectural registers, the L registers are also mapped to physical registers in a warp level. This design ensures that L registers are pro-actively released once their lifetime expires, in order to prevent an increase in register pressure. Unlike with R registers, the lifetime of the L registers depends not only on their parent thread but also on surrounding threads. The dead L registers are released once the last access to them occurs. The last access calculation depends on whether the register is accessed horizontally or vertically, or in both directions. The design takes care of this, as depicted in Fig. 19. Any register highlighted in orange is released after all threads that share it horizontally have

accessed it. Registers highlighted in blue are released after the threads that depends on them vertically have accessed them. Registers highlighted in red and grey are released immediately after their own parent thread has accessed them. Registers in red store weights, and registers in orange maintain the data that are not shared by other threads.

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Figure 19: Destination L register identifiers for input data and weights

7.3 Intra-Warp Sharing

The cross section of sharing depends on the convolutional kernel dimension, row stride, column stride, and skip factor, as shown in Fig. 20. These parameters should be given as compiler directives by the application developer. Due to this feature, the implementation is scalable for any CNN application. The skip factor is 2 in this case, as weights and input data are stored alternatively in the registers.

7.3.1 Horizontal Sharing

For kernel dimension 5 and column stride 2, the cross section of horizontal sharing is 3 columns, which comprises 15 (5*3) inputs. If the generated destination register of the load instruction is in the horizontal overlapped region, then the horizontal shift is computed to find the actual register of another thread that holds the data.

The input data are in an overlapped region if

$$Reg_{curr} \bmod (KernelDimensionX * Skipfactor) > (ColumnStride * Skipfactor)$$

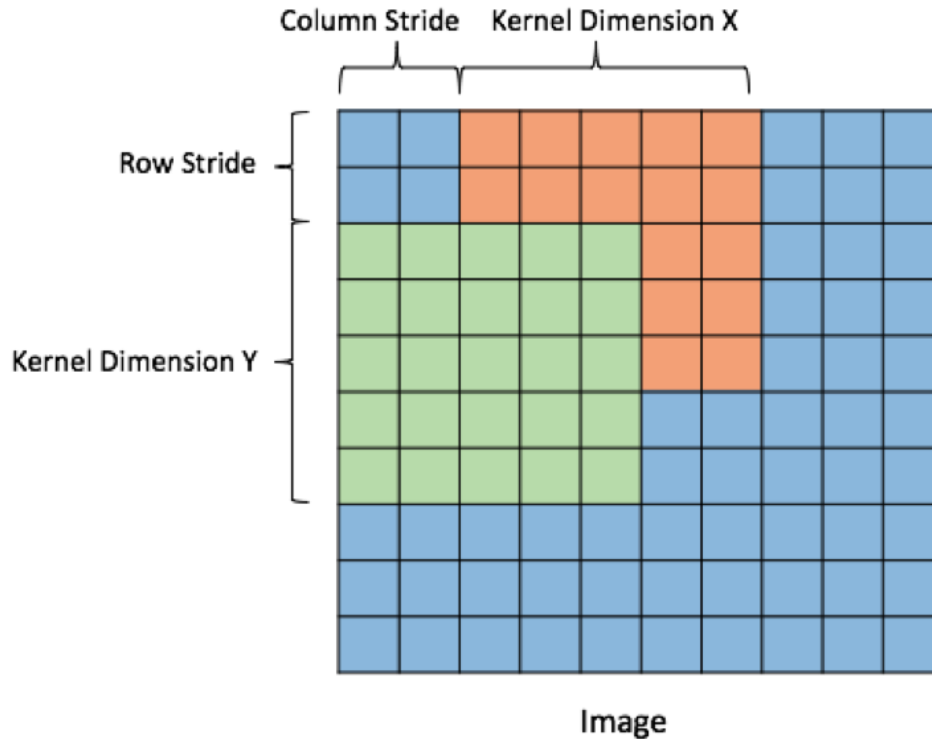

Figure 20: Convolutional kernel dimension and stride

Horizontal shift is given by

$$Reg_{to_be_reused} = Reg_{curr} - (ColumnStride * Skipfactor)$$

Then using shuffle instructions, this register can be accessed instead of the actual register.

7.3.2 Vertical Sharing

Vertical cross section depends on the kernel dimension X and the row stride of the kernel window. For kernel dimension 5 and row stride 2, the number of load instructions that do not overlap is 10 (kernel dimension * row stride). If the generated destination register is in the overlapped region, vertical shift is performed to calculate the register ID of the actual register that holds the data.

The input data are in an overlapped region if

$$Reg_{curr} > (KernelDimensionX * Skipfactor * RowStride)$$

Vertical shift is given by

$$Reg_{to_be_reused} = Reg_{curr} - (KernelDimensionX * RowStride)$$

7.4 New Active Mask for Overlapped Region

To skip the load instructions, the load-store instruction pipeline is interrupted. Generally, active masks are set before the warp is issued to the instruction pipeline. Active masks indicate which thread in a warp should execute an instruction. For the load instructions in the overlapped region, the active mask is nullified so that all threads of all warps skip those instructions. The load-store execution units remain idle while shuffle instructions are performed instead.

7.5 Handling Corner Cases

As explained in section 6.5, corner case threads skip the load instructions because there are no neighboring threads to fetch input data on their behalf. To address this, the design takes two approaches. In the first approach, the corner cases are handled by increasing the number of threads in the thread block, that is, by increasing the dimension of the thread block. In the second approach, the corner case is ignored and the quality of the neural network is measured by computing the error associated with each of the output neurons. Depending on the type of application as well as the accuracy or energy efficiency needs, an application can choose between the two.

7.5.1 Intra-Warp Boundary

The last thread in a thread block skips some of the load instructions with the L register implementation. However, these threads should not skip load instructions because data reuse is not possible at the boundary, as explained in section 6.5.1. To handle this, the current design proposes a simple solution. At the application level, the number of threads in the thread block is increased to execute the load instructions of the original boundary threads. With this simple approach, the original boundary threads can skip the load

instructions as per the L register implementation, because the new threads fetch the data on behalf of them. The new threads have the data in their L registers only for other threads to use them and not for themselves. Thanks to these new threads, corner case threads do not need to skip their computations. Therefore, the error rate doesn't need to be considered. The extended threads are just worker threads that help in the data fetch but do not perform any computations from that data. This idea is depicted in Fig. 21. The extra thread 6 fetches two columns of the input image and thread 7 fetches the last column. Thus, thread block dimension is increased from (X, Y) to (X+2, Y+2). This solution is applicable to all convolutional layers.

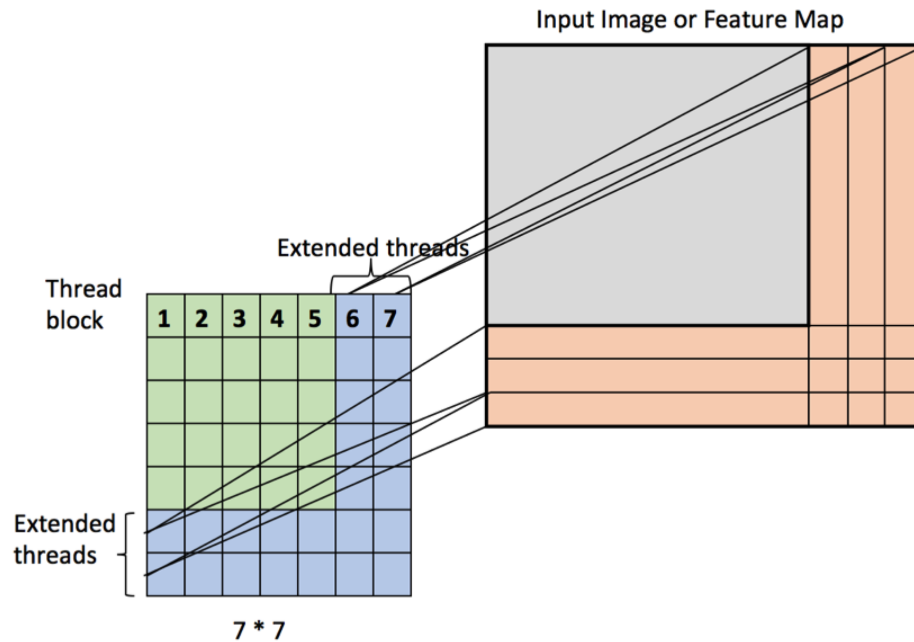

Figure 21: Solution for intra-warp boundary corner case thread

7.5.2 Inter-Warp Boundary

The design eliminates the horizontal inter-warp boundaries completely by avoiding the intersection of different warps. To make the design simple, the row dimension is increased by warp units such as a half warp (16 threads), a quarter warp (8 threads), a full warp (32 threads), or multiple warps as shown in Fig. 22. Thus, thread block dimension is increased

to (X+2+N, Y+2+N) where N is the number of threads that makes X+2+N a factor or multiple of warp unit. With this design, the inter-warp boundary vanishes. The additional threads are used just for making a better placement of threads in the warp and play no role in data fetching or computation.

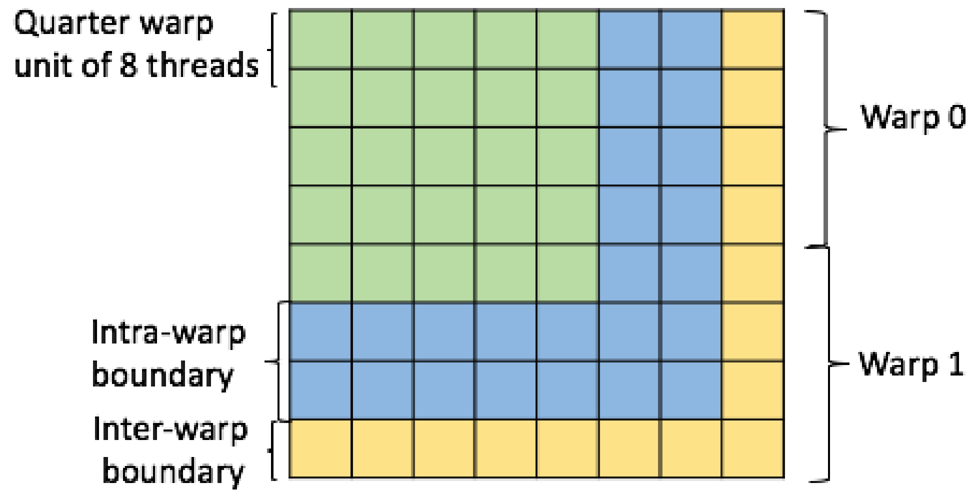

Figure 22: Solution for horizontal inter-warp corner case of boundary thread

CHAPTER 8

Experimental Results

8.1 GPGPU-Sim Setup and Modification

The L register design was implemented in GPGPU-Sim version 3.2.1 [41]. GPGPU-Sim is a cycle-level GPU architecture simulator. The simulator has two simulation modes: performance simulation to measure the timing and functional simulation to execute the instructions. The number of SMs for the simulator was set to 16. The size of the register file was set to 128 KB, that is, 32768 registers of 4 bytes each. The performance of the proposed design was compared with the design that used shared memory. To evaluate the performance impact of limited resources, the performance was measured with fewer SMs. Scheduler sensitivity of the proposed design was measured by using three schedulers, two-level scheduler, GTO scheduler, and round robin scheduler. For a realistic evaluation, the access latency of L1 cache and shared memory was varied from 10 cycles to 30 cycles [41]. Power was measured using GPUWattch [42].

8.2 Ignoring the Corner Case Threads

In this experiment, the corner cases of first and second convolutional layers were ignored. Fig. 23 shows the MSE for each layer of the four-layer CNN. The error rate was highest at layer 3, where a transition happened from convolution to fully connected layers. However, the error rate in the final output was negligible (about 0.1501). Overall, all layers produced a minimum deviation from normal, which is acceptable. All evaluations hereafter were done for two cases, with and without handling corner cases.

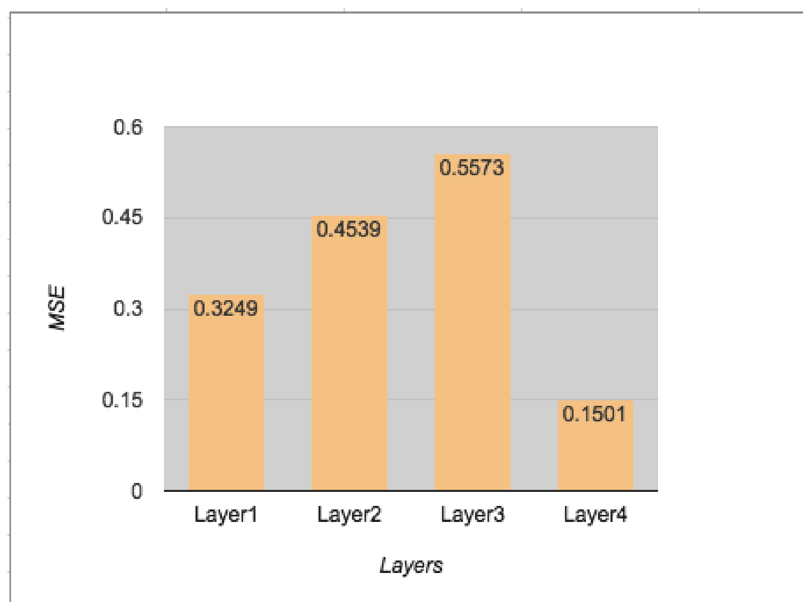

Figure 23: Mean square error of each layer after skipping corner cases

8.3 Performance

The performance was measured by normalizing the total simulation cycles of proposed design over the cycles of baseline implementation. As shown in Fig. 24, the proposed design, which is marked as LREG Opt, improved performance by 61% and 20% over baseline and the shared memory version, respectively.

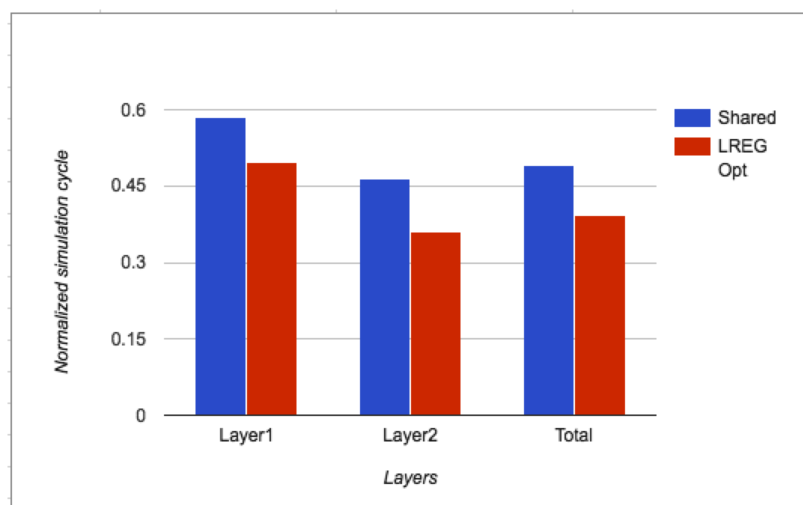

Figure 24: Comparison of performance of shared memory and L register without corner case

As shown in Fig. 25, the L register implementation with corner case delivered a 22.3% improvement over baseline, but there was a 30% drop compared to the shared memory version. This is because the L register used additional warps to handle the corner case threads, which increased the scheduling overhead. Although shared memory seems to have had a performance advantage in this case, it was applicable only for small image sizes. For larger images, the shared memory version was incapable of storing the images. On the other hand, the proposed L register implementation handled larger images by releasing registers appropriately. Since L registers are scalable for all image dimensions, the L register implementation is considered a better choice than the shared memory version.

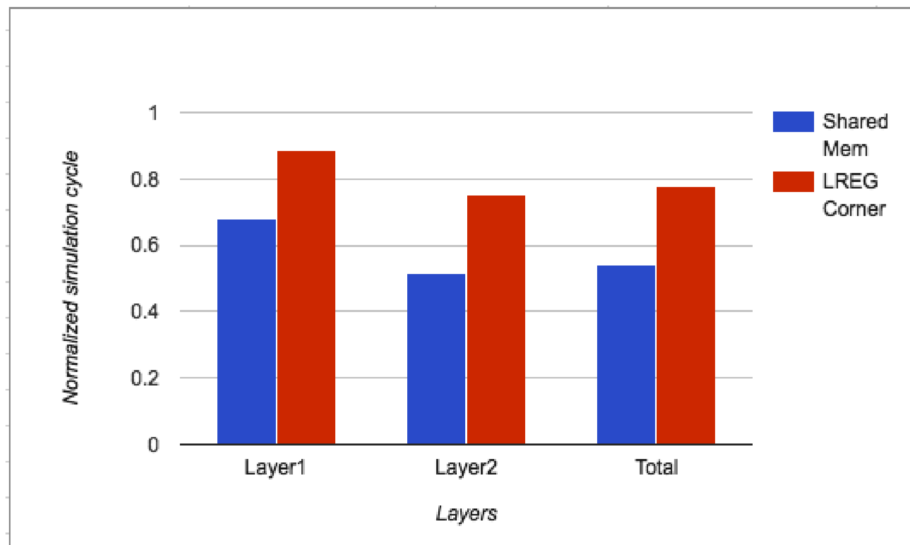

Figure 25: Comparison of performance of shared memory and L register with corner case

As plotted in Fig. 26, when fewer SMs were provided, the proposed design outperformed baseline and the shared memory version by 67% and 0.3%, respectively. For an MNIST 33*33 image and Cifar-10 image, as shown in Fig. 27 and Fig. 28, there was a 61% improvement over baseline and a 25% improvement over the shared memory version.

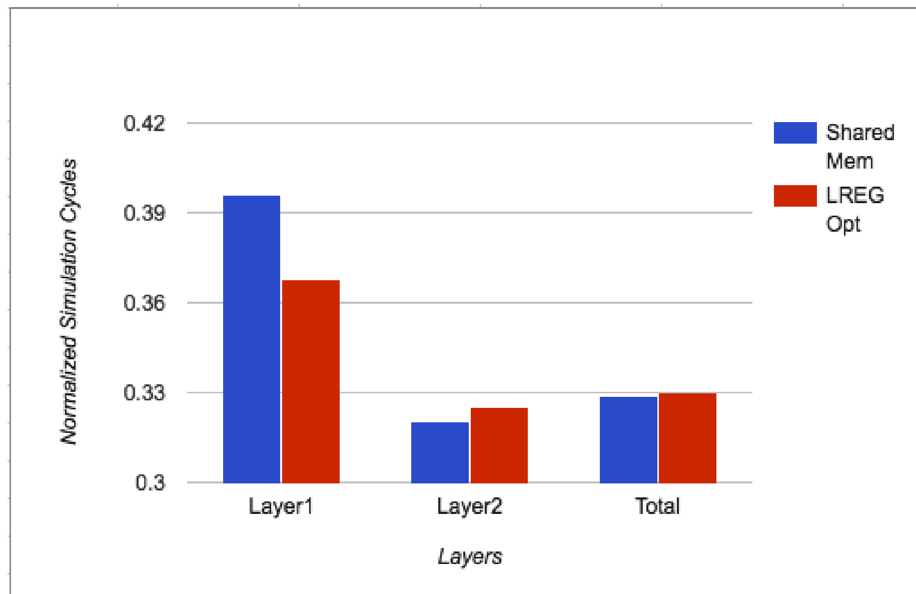

Figure 26: Comparison of performance of shared memory and L register without corner case using three clusters

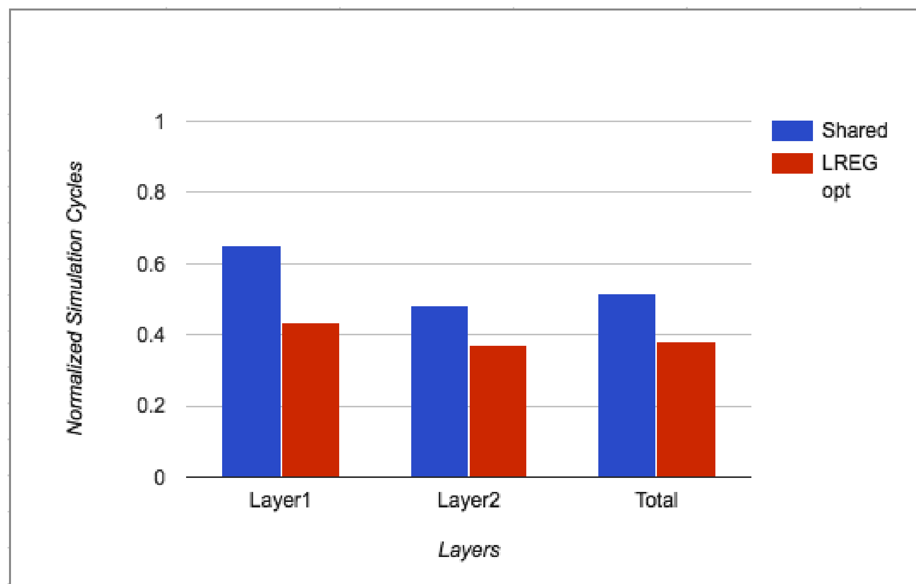

Figure 27: Comparison of performance of shared memory and L register with corner case for large image dimension of 33*33

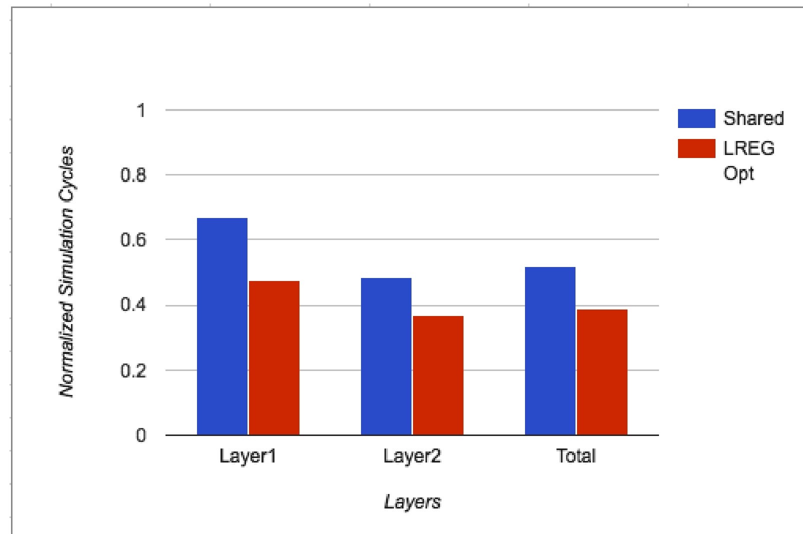

Figure 28: Comparison of performance of shared memory and L register with corner case for Cifar-10 input image

8.4 Load Instruction Reduction

The total number of executed load instructions was normalized by baseline. As depicted in Fig. 29, the L register-optimized implementation had a 42% reduction in the number of load instructions over baseline and a 23% reduction over the shared memory version.

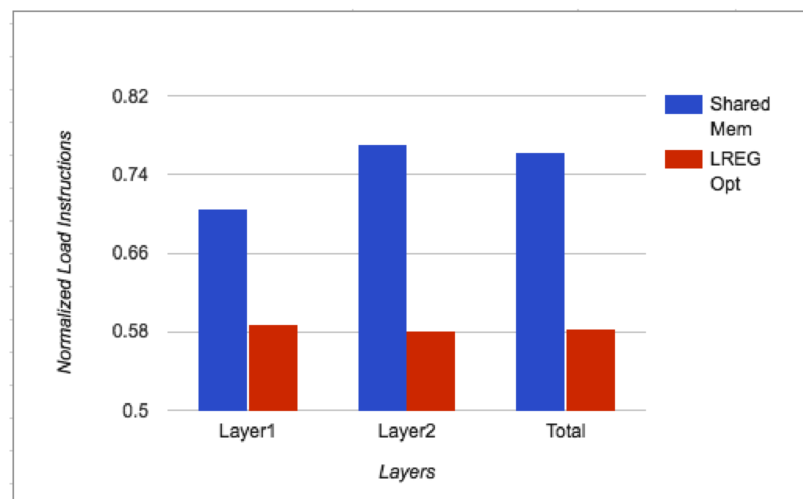

Figure 29: Comparison of load instruction count of shared memory and L register without corner case over baseline

As shown in Fig. 30, the L register implementation with corner case had a 39.3% reduction in the number of load instructions over baseline and an 18.9% reduction over the shared memory version. For MNIST with bigger image case (33*33), there was a 42% reduction from baseline and a 21% reduction over shared memory version as shown in Fig. 31.

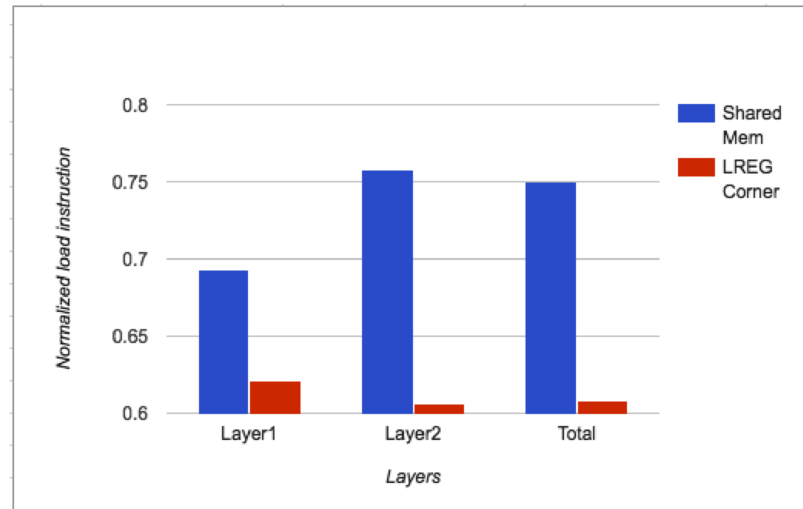

Figure 30: Comparison of load instruction count of shared memory and L register with corner case over baseline

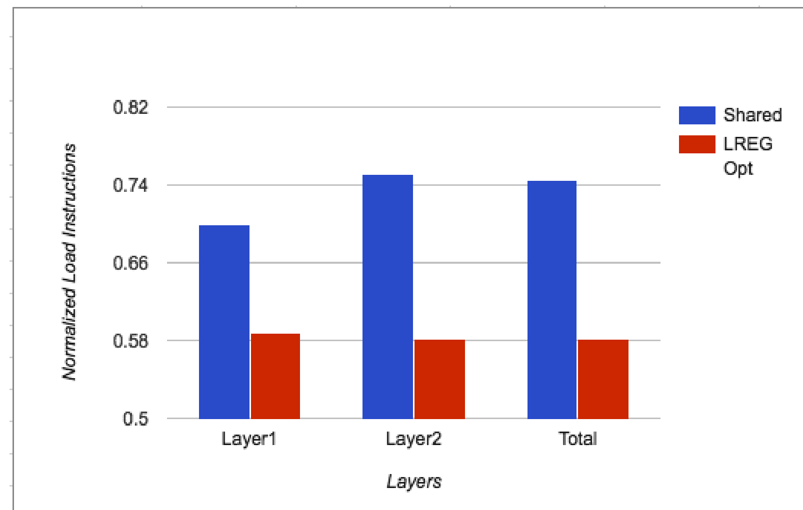

Figure 31: Comparison of load instruction count of shared memory and L register with corner case over baseline for large image dimension of 33*33

For the Cifar-10 dataset as plotted in Fig. 32, the reduction was similar to the reduction in MNIST larger image. Regardless of corner case handling, the L register implementation exhibited a notable reduction in the number of load instructions. Hence, it is a trade-off to choose between the two depending on whether MSE is acceptable or not. Since only input image data were shared, the weight values were still loaded from the global memory and included in the load instruction count.

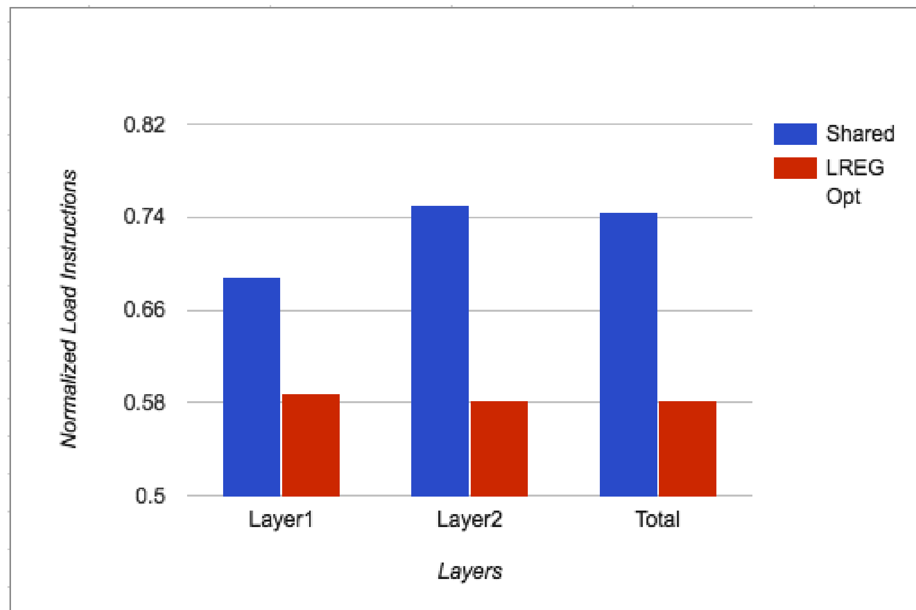

Figure 32: Comparison of load instruction count of shared memory and L register with corner case over baseline for Cifar-10 input image

8.5 Power Dissipation

When the load instructions are reduced, the load-store execution units can be shut down. This, in turn, reduces power consumption. The average power of L register without corner case was compared to baseline implementation in Fig. 33. Overall, L register power was 16.2% better than baseline. This is because global memory accesses were significantly prohibited in L register implementation.

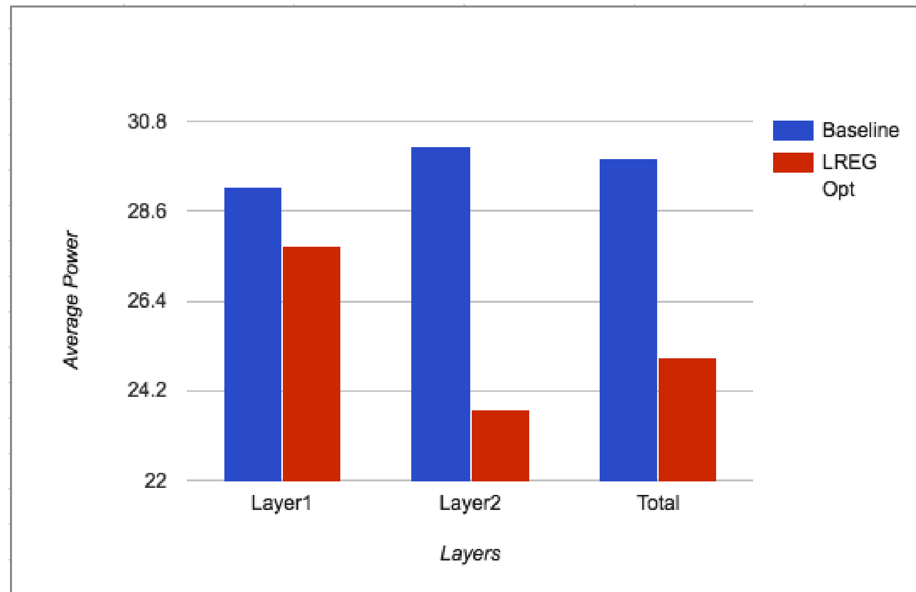

Figure 33: Comparison of average power dissipation of baseline and L register without corner case

8.6 On-Chip Memory Latency Sensitivity

The performance of the application varies with regard to latencies of L1 cache and shared memory. The latency sensitivity was measured for 10, 20, and 30-cycle access latencies for both L1 cache and shared memory. The simulation cycles were expressed in log scale for clearer interpretation of the results. As shown in Fig. 34, baseline implementation exhibited the highest sensitivity to latency with a drop of 65% IPC as latency elongates. The proposed design showed a 64% drop which proves that the L register design is less sensitive to latency variation. Shared memory implementation was better than baseline by 1.1%.

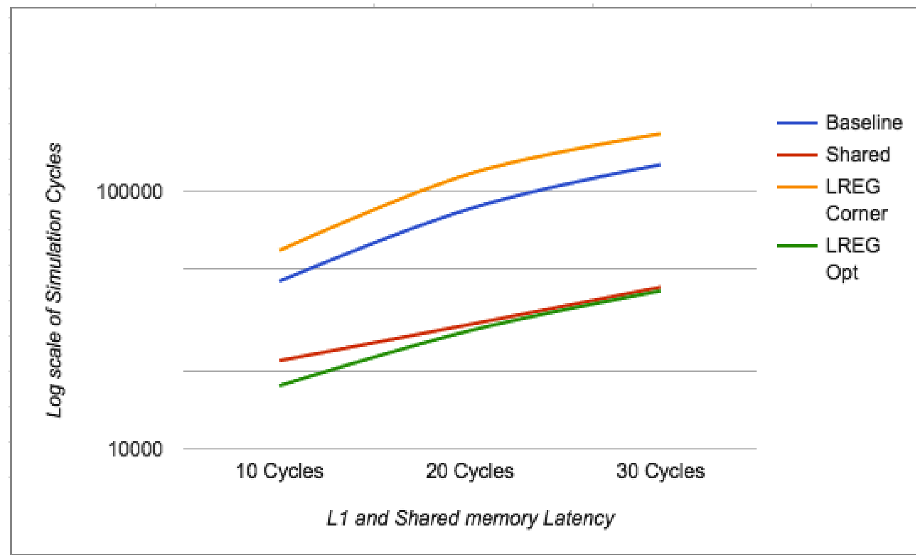

Figure 34: Comparison of latency sensitivity of baseline, shared memory, L register with corner case and L register without corner case

8.7 Scheduler Sensitivity

Fig. 35 shows the performance of L register for three types of schedulers. Two-level active scheduler [43] has two queues for the warps, an active and a pending queue. The active queue has warps that are ready to execute. The scheduler selects the warp from the active queue and issues it to the pipeline for execution. All warps executing pending instructions with long latencies are moved to the pending queue. Once a warp becomes ready it is moved from the pending queue to the active queue. Greedy then oldest scheduler (GTO) aims to increase the occupancy by distributing thread blocks to an SM whenever an SM has hardware resources to support execution. Loose round-robin scheduler (LRR) assigns equal priority to all the warps and therefore, schedules warps on a first come first serve basis. LRR scheduler was used as baseline for the comparison. TLA scheduler's overall performance was similar to that of LRR. GTO scheduler had a 1% performance degradation compared to LRR and TLA.

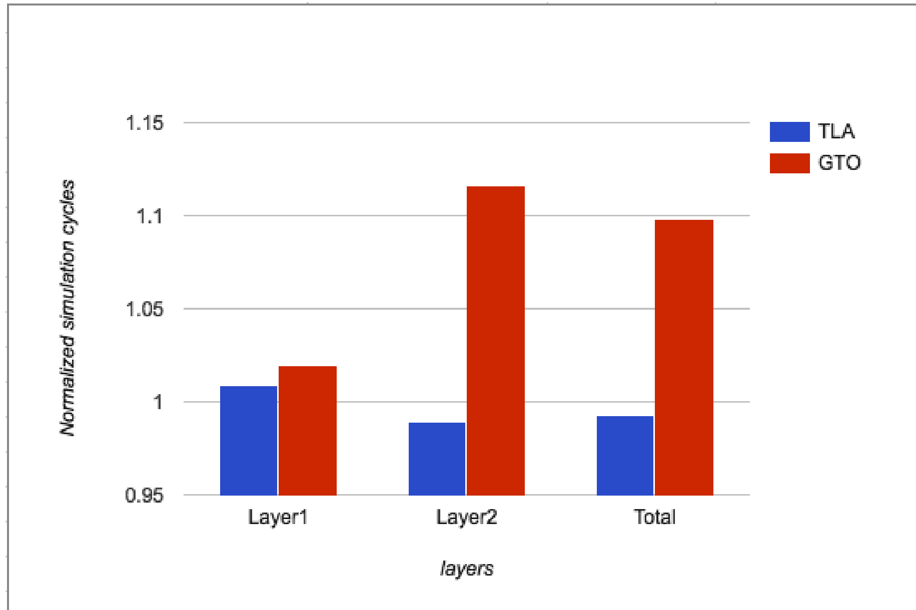

Figure 35: Scheduler sensitivity of L register for TLA and GTO warp schedulers normalized over LRR scheduler

CHAPTER 9

Future Work

The current L register scheme applies only to convolutional layers of DNN. Next to convolutional layers, fully connected layers are computationally intensive. Fully connected layers handle a huge chunk of input data, but with slightly different access patterns. Therefore, an extension to fully connected layers will improve the performance even more. Vertical inter-warp sharing was not handled in this thesis due to the limited support for the data exchange across warps in current GPU design. The proposed efficient register decoder scheme could help in crossing the warp boundaries. Evaluations with deeper and popular networks like AlexNet [7], VGG-16 [17], and GoogleNet [18] would give valuable insights on the performance of L register implementation. However, these networks require significant adaptations to be ready for testing on the simulator. Currently, I am working on creating similar deep neural network applications.

CHAPTER 10

Conclusion

The default GPU implementation does not efficiently reuse the input data of DNN applications. Therefore, there are excessively redundant data fetches from slower memories, leading to a significant performance overhead. L register implementation can resolve this problem with an efficient register file allocation and management mechanism. The hardware uses the compiler information to identify the destination registers that need to retain data. The large input data of NN applications are kept close to the GPU and reused as much as possible. According to the evaluation results, data movement from global memory was reduced and performance was improved, resulting in a substantially smaller memory footprint.

References

[1] Y. LeCun, K. Kavukcuoglu and C. Farabet, "Convolutional Networks and Applications in Vision," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2010, pp. 253-256.

[2] L.L.C. Kasun, H. Zhou, G. Huang and C.M. Vong, "Representational learning with ELMs for big data," in *IEEE Intelligent Systems*, 2013, vol. 28, pp. 31-34.

[3] D.B. Kirk and W.H. Wen-Mei, "Programming Massively Parallel Processors: A Hands-on Approach," Morgan Kaufmann, 2016.

[4] NVIDIA. (2009). *Fermi:Nvidia's next generation cuda compute architecture* [Online]. Available: http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_ _compute_architecture_whitepaper.pdf [Feb. 15, 2017].

[5] NVIDIA. *Nvidia's next generation cuda compute architecture: Kepler gk110/210 version 1.1. Apress* [Online]. Available: https://www.microway.com/download/ whitepaper/NVIDIA_Kepler_GK110_GK210_Architecture_Whitepaper.pdf [Feb. 15, 2017].

[6] N.G. GTX. (2014). *980: Featuring Maxwell, the most advanced GPU ever made* [Online]. Available: https://www.microway.com/download/ whitepaper/NVIDIA_Maxwell_GM204_Architecture_Whitepaper.pdf [Feb. 15, 2017].

[7] A. Krizhevsky, I. Sutskever and G.E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, 2012, pp. 1097-1105.

[8] Y. LeCun, Y. Bengio and G. Hinton, "Deep learning," *Nature*, 2015, vol. 521, pp. 436-444.

[9] M. Mohri, A. Rostamizadeh and A. Talwalkar. "Foundations of Machine Learning," MIT Press, 2012.

[10] D. E. Rumelhart, G. E. Hinton, R. J. Williams, "Learning Internal Representations by Error Propagation," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. Vol. 1: Foundations, MIT Press, 1986.

[11] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *International Conference on Artificial Neural Networks*, pp. 281-290, 2014.

[12] S.W. Keckler, W.J. Dally, B. Khailany, M. Garland and D. Glasco, "GPUs and the future of parallel computing," *IEEE Micro*, vol. 31, pp. 7-17, 2011.

[13] M.D. McCool, A.D. Robison and J. Reinders, "Structured Parallel Programming: Patterns for Efficient Computation," Elsevier, 2012.

[14] D. Kirk, "NVIDIA CUDA Software and GPU Parallel Computing Architecture," in *International Symposium on Memory Management*, pp. 103-104, 2007.

[15] NVIDIA. (2011). *Nvidia cuda c programming guide* [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html [Feb. 27, 2017].

[16] K. He, X. Zhang, S. Ren and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770-778.

[17] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2006, arXiv Preprint arXiv:1409.1556.

[18] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke and A.Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 1-9.

[19] D. Jacobs. (2005). *Correlation and convolution* [Online]. Available: http://www.cs.umd.edu/ djacobs/CMSC426/Convolution.pdf [Feb. 27, 2017].

[20] M. D. Zeiler and R. Fergus, "Visualizing and Understanding Convolutional Networks," in *European Conference on Computer Vision*, Springer International Publishing, Zurich, Switzerland, 2014, pp. 818-833.

[21] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G.S. Corrado, A. Davis, J. Dean and M. Devin, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," 2016, arXiv Preprint arXiv:1603.04467.

[22] NVIDIA. *NVIDIA System Management Interface* [Online]. Available: https://developer.nvidia.com/nvidia-system-management-interface [Mar. 2, 2017].

[23] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM International Conference on Multimedia*, 2014, pp. 675-678.

[24] S. Sonnenburg. (2006). *Nips workshop on machine learning open source software* [Online]. Available: http://www2.fml.tuebingen.mpg.de/raetsch/ workshops/MLOSS06/ [Mar. 2, 2017].

[25] NVIDIA Corporation. (2014). *Cuda profiler users guide (version 6.5): Nvidia* [Online]. Available: http://docs.nvidia.com/cuda/profiler-users-guide/ [Mar. 10, 2017].

[26] L. Yu, Y. Pei, T. Chen and M. Wu, "Architecture supported register stash for GPGPU," Journal of Parallel and Distributed Computing, 2016, vol. 89, pp. 25-36.

[27] S. Mittal, "A survey of techniques for architecting and managing GPU register file," *IEEE Transactions on Parallel and Distributed Systems*, 2017, vol. 28, pp. 16-28.

[28] H. Jeon, G.S. Ravi, N.S. Kim and M. Annavaram, "GPU Register File Virtualization," in *Proceedings of the 48th International Symposium on Microarchitecture,* 2015, pp. 420-432.

[29] S. Lee, K. Kim, G. Koo, H. Jeon, M. Annavaram and W.W. Ro, "Improving energy efficiency of gpus through data compression and compressed execution," *IEEE Transactions on Computers*, 2016.

[30] N. Alsaber and M. Kulkarni, "Semcache: Semantics-aware caching for efficient gpu offloading," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, 2013, pp. 421-432.

[31] H. Wu, G. Diamos, J. Wang, S. Cadambi, S. Yalamanchili and S. Chakradhar, "Optimizing data warehousing applications for GPUs using kernel fusion/fission," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW)*, IEEE 26th International Conference, 2012, pp. 2433-2442.

[32] S. Wang, L. Kan, Y. Hwang and J. Lee, "Energy efficient affine register file for GPU microarchitecture," in *45th International Conference on Parallel Processing Workshops (ICPPW)*, 2016, pp. 52-58.

[33] X. Xie, Y. Liang, X. Li, Y. Wu, G. Sun, T. Wang and D. Fan, "Enabling coordinated register allocation and thread-level parallelism optimization for GPUs," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 395-406.

[34] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen and T. Chen, "Cambricon: An instruction set architecture for neural networks," in *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016, pp. 393-405.

[35] X. Yang, J. Pu, B.B. Rister, N. Bhagdikar, S. Richardson, S. Kvatinsky, J. Ragan-Kelley, A. Pedram and M. Horowitz, "A systematic approach to blocking convolutional neural networks," 2016, arXiv Preprint arXiv:1606.04209.

[36] J. E. Y. Chen and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 367-379.

[37] A. Bakhoda, G.L. Yuan, W.W. Fung, H. Wong and T.M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *IEEE International Symposium on Performance Analysis of Systems and Software*, 2009, pp. 163-174.

[38] Y. LeCun, C. Cortes and C.J. Burges. (1988). *The MNIST database of handwritten digits* [Online]. Available: http://yann.lecun.com/exdb/mnist/ [Mar. 14, 2017].

[39] P. Collingbourne, A.F. Donaldson, J. Ketema and S. Qadeer, "Interleaving and Lock-Step Semantics for Analysis and Verification of GPU Kernels," *European Symposium on Programming*, 2013, pp. 270-289.

[40] G. Casella and E.L. Lehmann, "Theory of Point Estimation," New York: Springer-Verlag, 1999.

[41] T. Aaamodt and A. Boktor. (2012) *GPGPU-Sim 3.x: A performance simulator for many core accelerator research* in *International Symposium on Computer Architecture (ISCA)*, [Online]. Available: http://www.Gpgpu-Sim.org/isca2012-Tutorial [Mar. 16, 2017].

[42] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N.S. Kim, T.M. Aamodt and V.J. Reddi, "GPUWattch: Enabling energy optimizations in GPGPUs," in *ACM SIGARCH Computer Architecture News*, vol. 41, 2013, pp. 487-498.

[43] V. Narasiman, M. Shebanow, C.J. Lee, R. Miftakhutdinov, O. Mutlu and Y.N. Patt, "Improving GPU performance via large warps and two-level warp scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 308-317.