Characterizing the Microarchitectural Implications of a Convolutional Neural Network (CNN) Execution on GPUs

Conference Paper · April 2018

DOI: 10.1145/3184407.3184423

CITATIONS
11

READS
1,080

5 authors, including:

Shi Dong
Northeastern University
15 PUBLICATIONS **70** CITATIONS
SEE PROFILE

Xiang Gong
Northeastern University
11 PUBLICATIONS **90** CITATIONS
SEE PROFILE

Yifan Sun
William & Mary
30 PUBLICATIONS **210** CITATIONS
SEE PROFILE

David Kaeli
Northeastern University
443 PUBLICATIONS **4,504** CITATIONS
SEE PROFILE

Characterizing the Microarchitectural Implications of a Convolutional Neural Network (CNN) Execution on GPUs

Shi Dong
Dept. of Electrical and Computer Engineering
Northeastern University
shidong@ece.neu.edu

Xiang Gong
Dept. of Electrical and Computer Engineering
Northeastern University
xgong@ece.neu.edu

Yifan Sun
Dept. of Electrical and Computer Engineering
Northeastern University
yifansun@ece.neu.edu

Trinayan Baruah
Dept. of Electrical and Computer Engineering
Northeastern University
tbaruah@ece.neu.edu

David Kaeli
Dept. of Electrical and Computer Engineering
Northeastern University
kaeli@ece.neu.edu

ABSTRACT

GPUs have become a very popular platform for accelerating the processing involved in deep learning applications. One class of popular variants, Convolutional Neural Networks (CNNs), have been widely deployed to run on GPUs. In many application settings, a GPU has sufficient computing power and memory space to accommodate the dense matrix operations performed during CNN training. However, few characterization studies have considered how CNNs can impact microarchitectural structures in a GPU. In this paper, we perform a characterization of one selected CNN workload as run on two different NVIDIA GPUs from distinct microarchitecture families, highlighting the impact that microarchitecture plays on this important class of workload.

First, we analyze the performance implications of a CNN model using microarchitectural details on a layer-by-layer basis, and characterize the memory access behavior in the context of a typical GPU memory hierarchy, considering hardware resource utilization associated with each primitive in the CNN model. We identify major bottlenecks by considering the potential limits of using a single GPU. Additionally, we evaluate a number of optimization approaches, such as L1 cache bypassing and kernel fusion. L1 cache bypassing can achieve up to a 6.2% speedup for a single layer, but manipulating L1 cache provides very limited benefits in terms of application speedup, while kernel fusion provides an overall application speedup of 4.0%, on average.

CCS CONCEPTS

• **General and reference** → **Evaluation**; **Performance**; • **Computer systems organization** → **Neural networks**; • **Computing methodologies** → *Neural networks*;

KEYWORDS

Convolutional Neural Networks; GPU; Characterization; Performance analysis

ACM Reference Format:
Shi Dong, Xiang Gong, Yifan Sun, Trinayan Baruah, and David Kaeli. 2018. Characterizing the Microarchitectural Implications of a Convolutional Neural Network (CNN) Execution on GPUs. In *Proceedings of ACM/SPEC International Conference on Performance Engineering, Berlin, Germany, April 9–13, 2018 (ICPE '18),* 11 pages.
https://doi.org/10.1145/3184407.3184423

ICPE '18, April 9–13, 2018, Berlin, Germany

ACM ISBN 978-1-4503-5095-2/18/04…$15.00
https://doi.org/10.1145/3184407.3184423

1 INTRODUCTION

Deep Learning (DL) algorithms have emerged as a "celebrity" in the field of machine learning, especially given that they can leverage GPUs to accelerate their execution. Deep Learning and Artificial Neural Networks were first described many years ago, but were considered viable for learning problems until recently. With support of the state-of-the-art GPU hardware, neural networks have been deployed in a wide range of applications including artificial intelligence, natural language processing, and human-computer interfacing. A commonly cited exemplary application is autonomous vehicle guidance [8]. Using DL algorithms, a self-driving car is able to identify its surrounding environment and make corresponding movements without any human intervention. Another highly publicized AI-related application, AlphaGO, uses DL as one of its key algorithms during the training process [25]. In order to capture the rapid growth of deep learning research, a number of frameworks have been developed by both the academic and industry communities to further facilitate their development. Caffe [9], TensorFlow[1], Theano [26], CNTK [22] and MXNet [2] are a few of the well-known deep learning frameworks that provide users to design and deploy neural network models efficiently.

Convolutional Neural Network (CNN) is one popular variant of deep neural network (DNN) that leverages convolution as its major linear transformation for feature extraction. It has been demonstrated that CNN can be very effective in vision and speech classification domains [11, 13]. Similar to other models of deep neural

networks, the essential computations in CNNs have been accelerated using a GPU, especially given that most computational operations involving convolution are matrix-based. Since GPUs are an effective target for accelerating matrix operations, they have been quickly developed into a key resource for accelerating CNNs. However, there has only been limited prior work with regards to the execution of this type of workload on GPUs. Given this limited knowledge, it becomes challenging to optimize GPU architectures to run this class of compute-intensive applications.

In this paper, we capture and analyze the micro-architectural information from two GPUs. The two GPUs are of different product grade, server (Tesla K40) and desktop (GTX1080), and from different microarchitecture family, kepler and pascal, respectively. After then, we study the microarchitectural implications of efficiently running CNNs. To begin this study, we have selected to study AlexNet [11], which is a popular CNN model. Even though the AlexNet is not the latest state-of-the-art CNN model, it covers most of the commonly used primitives in deep neural networks, and most importantly, its structure is simple to evaluate and its execution presents a number of challenges to current GPU designs. In terms of an implementation of AlexNet, we utilize DNNMark [7] to drive this study, providing a highly configurable and light weight infrastructure, that builds its core function using cuDNN[3] and cuBLAS[5]. These two highly optimized libraries have been used in many DNN frameworks that leverage Nvidia GPUs. Considering their performance, cuDNN and cuBLAS provide a rich software core for us to study DNN execution, working at a microarchitectural level. In our evaluation, we report on execution performance, memory behavior, and resource utilization. Furthermore, we identify some of the major limiting factors in GPU microarchitectures when executing DNN primitives.

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

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

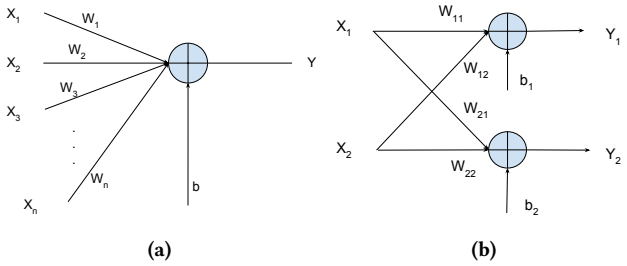


Figure 1: (a) Model of an artificial neuron. (b) A simple example of a fully-connected layer.

The rest of this paper is organized as follows. In section 2, we provide an overview of the basic elements and structure of a convolutional neural network, and review GPU architecture. In section 3, we discuss our characterization approach, the database-backed trace tracking framework and the details of the hardware used for experiments. In section 4, we present detailed analysis on both an entire model and individual primitives. In section 5 we review the contributions of this work, and in section 6 we discuss previous work on convolutional neural networks characterization, and in section 7 we conclude the paper.

2 BACKGROUND

2.1 Deep Neural Networks

The operations involved in Deep Neural Networks are basically combinations of linear and non-linear data transformations, as well as other techniques used to avoid over-fitting and improve prediction accuracy. In this paper, we mainly concentrate on Convolutional Neural Network (CNN), a variant of a DNN that applies convolutional operations. Generally, in a CNN model, the linear data transformations can be broken down into two major categories: i) fully-connected and ii) convolution. These two transformations are carried out by applying elementary operations, such as multiplications and summations, which are performed on data, trainable weights, and biases. For image processing, the data moving across the network model is usually managed in a tensor format, meaning that the data has at least 4 dimensions, N, C, H, and W, in which N is the batch size, C is the number of channels, H is the height, and W is the width. The tensor data is stored in memory as a matrix in column-major format, with the number of rows equal to $C * H * W$ and the number of columns equal to N.

2.1.1 Linear Data Transformations. In the **fully-connected layer**, the basic unit to construct is an artificial neuron, which is a simple mathematical model, as presented in Figure 1a. The neuron is composed of operations such as multiplications and summations, as expressed in Equation 1.

$$Y = \sum_{i=1}^{n} W_i * X_i + b \quad (1)$$

The fully-connected layer is constructed of multiple artificial neural interconnects, as shown in figure 1b.

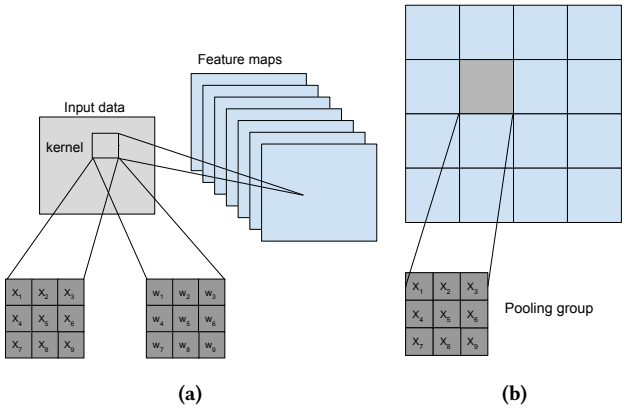


Figure 2: (a) 2-D image convolution. (b) Feature map data divided into multiple pooling groups.

Based on this model, the fully-connected layer can be interpreted using a matrix-vector multiplication, as indicated in equation 2. This example has only one set of data inputs. If there are multiple sets of data inputs, then the computation becomes a multiplication and a summation of matrices.

$$\begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix} = \begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \end{bmatrix} * \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{2}$$

In the **convolution layer**, the operations are performed in a different fashion. Figure 2a shows an example of a 2-D image convolution that is used in a typical CNN model.

In this figure, a convolution kernel of 3X3 is applied to an area in the image of the same size. The input data is X_i, and the kernel weights are W_i. Using the computation shown in equation 3, we can calculate a single result for Y based on the data covered by the kernel as follows:

$$\mathrm{Y_{i,j}} = \sum_{h=-k_h/2}^{k_h/2} \sum_{w=-k_w/2}^{k_w/2} \mathrm{W_{h,w} * X_{i+h,j+w}} \tag{3}$$

In the above equations, k is the kernel size. In the next step, the kernel is applied to the next data sample using a specified stride, and we repeat the same process across samples of every channel until all of the feature maps are calculated. During image convolution, the weights are shared by all of the data samples (i.e., pixels, if image data is used). Note that, as the stride grows, the calculated feature map has a narrower height and width.

There are multiple options on how best to perform 2D image convolution, including Direct, GEMM, FFT [20], and Winograd [12]. The Direct algorithm expresses the convolution as a direct convolution [3]; The GEMM method transforms the entire process into a matrix-matrix multiplication. The FFT and Winograd algorithms are fast implementations that are widely used [12], the former requires significant memory space, while the latter is memory-efficient [6].

2.1.2 Non-linearity. The non-linearity is introduced as **activation functions** to deal with linear inseparable problems. Some commonly used activation functions include the Rectified Linear Unit (ReLU), the sigmoid, and the hyperbolic tangent (tanh) [19].

The non-linear activation functions should be used together with linear transforms, meaning that every activation function follows a linear data transformation. The equations below provide mathematical descriptions of these functions.

$$\mathrm{y_i} = \max(0, x_i) \qquad \mathrm{y_i} = \frac{1}{1+e^{-x_i}} \qquad \mathrm{y_i} = \frac{e^{x_i} - e^{-x_i}}{e^{x_i} + e^{-x_i}} \tag{4}$$

These functions are implemented as element-wise matrix operations.

2.1.3 Other Techniques. Some other techniques involved in neural networks computations include **pooling**, **local response normalization (LRN)**, and **Softmax**. Each one of these steps has a different mathematical model, and each with its own functionality and operations. Pooling is a down-sampling technique to reduce the amount of computation for the following layers, and it has been shown to be an effective approach to improve robustness in practice. As described in the Alexnet documentation [11], the pooling layer can reduce the error rate by around 0.4%. Normally, pooling groups are not overlapped, so the pooling layer can be viewed as a grid of pooling groups spaced k data samples apart [11], where k is the size of the pooling group. Figure 2b shows an example of the feature map data being divided into 16 pooling groups.

The down sampling within one pooling group can be completed either by selecting the maximum value (i.e., Max Pooling) or computing the average of the group (i.e., Average Pooling). Max Pooling is usually more widely used in CNN models, e.g. AlexNet. Max Pooling selects the most representative value from a sub-group of the feature map. By doing so, interference by neighboring pixels can be reduced significantly. Similar to convolution, pooling also needs a kernel to select the pooling group. But the kernel cannot be overlapped when it moves to next data sample. Besides, we can see that Max Pooling has no complex arithmetic operations executed, other than simple comparison and loading/storing of the data. The only computation is to identify the maximum value among the pooling groups.

In general, LRN is a normalization method that works across various feature maps or channels. Basically, normalization is done across data samples from multiple adjacent kernel maps, but at the same relative position. Equation 5 describes its computation:

$$\mathrm{y_i} = \frac{x_i}{k + \alpha(\sum_{j=\max(0,\, i-n/2)}^{\min(N-1,\, i+n/2)} (\mathrm{x_j})^2)^\beta} \tag{5}$$

where k, α, β are all configurable parameters of LRN, N is the number of kernel maps, n is the window size for normalization, x_i is the input data, and y_i is the output with the same spatial location. With this algorithm, the prediction error rate can drop by 2%[11].

The computation of the Softmax function should be performed at the end of neural network model. It is the core function in the output layer, as it interprets the output data from the previous layer, and generates a set of probability-like values in the range of 0 to 1 (note, the sum of all of these values equals 1). For each value that Softmax computes, it represents the extent to which the input data should be classified into one of the predefined classes. The Softmax function is defined in equation 6:

$$Y_i = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}} \quad (6)$$

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

2.1.4 Training of Neural Networks. To describe the entire training process mathematically, we can treat the whole network model as a loss function, specifying inputs, outputs and model parameters as arguments. The objective is try to optimize the in-network parameters so that the overall loss can be minimized.

Training deep neural networks is an iterative and time consuming task. It usually requires more than a thousand iterations before the network parameters are properly trained. One of the most effective algorithms used to update the parameters is *Stochastic Gradient Descent* (SGD), which is an iterative algorithm that processes a mini-batch of the training data [11].

The detailed mechanism for training can be further divided into a forward and a backward propagation. The purpose for performing forward propagation during training is to calculate the loss of the overall network based on the current parameters. The ultimate goal of the backward propagation is to obtain the derivatives of the loss function, with respect to the parameters for the SGD algorithm. When computing the loss, we are able to calculate the derivatives with respect to the inputs, outputs, and parameters of each layer by applying the derivative chain rule in a backward-cascaded fashion.

2.2 Graphic Processing Units

The architecture of a Graphic Processing Unit (GPU) is designed to improve instruction throughput rather than reduce the latency of a single instruction. As such, the compute unit organization is much simpler than a CPU core design. Nevertheless, the GPU has many more cores than a CPU and is able to run thousands of threads simultaneously with support of low-overhead thread switching to hide latency. Therefore, GPUs clearly outperform CPUs in most cases where instruction throughput matters. A good example is with matrix-based computations, which are heavily used in almost every DNN variant. Thus, GPUs are well-suited to execute DNNs that are designed to run on parallel platforms.

The basic building block of an Nvidia GPU is the multi-threaded Streaming Multiprosessors (SMs). As shown in Figure 3, each SM contains a collection of computational resources that includes single precision CUDA cores, load store units, special functional units, and texture units. Each SM is also equipped with a large register file, so that threads can have their own set. Dedicated registers, assigned to each thread, means that data no longer needs to be swapped out during context switching, as adopted in CPU This can potentially reduce the corresponding overhead. With a low context switch overhead, threads can hide pipeline stalls and effectively utilize the computational resources of the GPU.

An array of SMs is connected to a hierarchical memory system. Each SM has limited memory resources that are exclusive to themselves, including the L1 cache, the shared memory and the read-only/texture cache. The shared memory is a scratchpad cache that is accessible by the programmer. In Kepler architecture, it shares a configurable on-chip memory area with the L1 cache, whereas it is a dedicated memory in Pascal. The read-only/texture

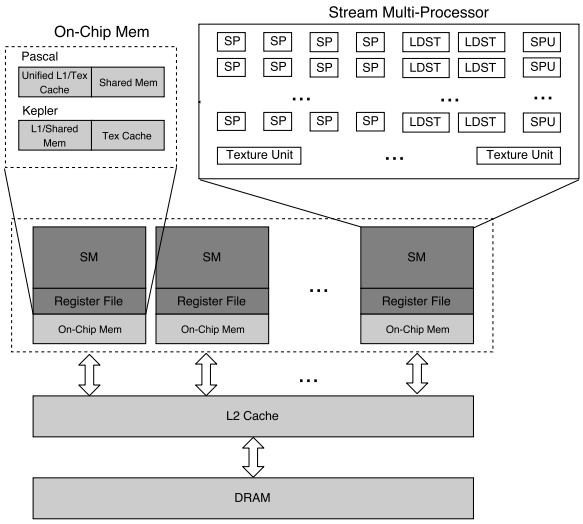


Figure 3: Diagram of a typical GPU architecture.

cache is accessible by the texture unit and the SM for general load operations. It is a dedicated memory in Kepler but shares a configurable on-chip memory area with the L1 cache in Pascal. Each SM has exclusive memory resources connected to a shared L2 cache. The L2 cache is the primary point of data unification between the SMs, servicing all general and texture memory requests, providing efficient, high speed data sharing across the GPU. The L2 cache is backed by a high bandwidth DRAM memory, which is the largest memory on a GPU that programmers can access directly.

An SM in a Pascal GPU has 128 single-precision (FP32) CUDA cores. In comparison, in the Kepler family, each SM has 192 CUDA cores. Pascal maintains the same register file size and supports similar occupancy of warps and thread blocks. However, more SMs can be accommodated in a Pascal GPU thanks to the smaller SM and better technology. Overall, the size of shared memory on the Pascal GPU is also increased due to the increased SM count, and aggregate shared memory bandwidth is effectively improved. A higher ratio of shared memory, registers, and warps per SM in Pascal GPUs allows the SM to execute code more efficiently. An SM in Pascal also features a simpler datapath organization that requires less die area and power to manage data transfers within the SM. Pascal also provides superior scheduling and overlapped load/store instructions to increase floating point utilization.

3 EVALUATION METHODOLOGY

3.1 Workload

In this paper, we select the AlexNet model to drive our characterization study. Although it is not the latest CNN model, it provides an organization that lends itself to evaluation, while including almost all of the primitives widely used in current state-of-the-art CNN models. Therefore, our microarchitectural characterization when running Alexnet can serve as a representative CNN model. Figure 4 shows the organization of AlexNet. As shown in this figure, AlexNet consists of 5 convolution layers, 3 fully-connected layers, 3 maxpooling layers, 2 LRN layers, 7 ReLU activation layers, and 1

Softmax layer. The number of operations in each layer is listed in Table 1. Note that we count every occurrence of either arithmetic or logical operations.

In terms of workload, we select cuDNNv6[3], a highly optimized DNN library specifically designed to run on Nvidia GPUs. This implementation has been used extensively by the deep learning research community, since it provides a user-friendly interface and is able to achieve high performance in terms of execution time. We use DNNMark [7], a configurable DNN benchmark suite composed of both cuDNN and cuBLAS, to construct the AlexNet model. Unlike applications found in other popular DNN frameworks, the AlexNet benchmark constructed within DNNMark is designed specifically for measuring hardware performance, essentially reducing the benchmarking effort by removing the need to develop new code. For input, we use a set of synthetic images, generated in batches, with the same dimensions as shown in the figure 4.

Table 1: Number of operations in each layer of AlexNet.

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

3.2 Hardware

In this paper, we select the Nvidia Tesla K40 [18] and GTX1080 [16] as the hardware platforms to run our experiments. The K40 microarchitecture was developed as part of Nvidia Kepler family of GPUs [14], while the GTX1080 is part of the Pascal family. They represents different product grades, as well. K40 is designed for servers, while the GTX1080 is designed for desktop acceleration. These two platforms have different architectures and computing capabilities, serving as good candidates to capture performance trends, while migrating the same workload from one platform to another. Table 2 provides details about each device.

3.3 Profiling Tools

Capturing and parsing the micro-architectural information of CNNs has many challenges due to the limitations of Nvidia profiler, nvprof[15]. These issues include: i) some layers invoke the same GPU kernel, but specify very different kernel template arguments, meaning that even though the kernel names are identical, they are in fact different kernels. If we want to capture the layer-specific information, using the kernel name only is not sufficient to uniquely identify the kernel; ii) some layers launch the exact same kernel based on their invocation order, according to the network model, so profiling using only the kernel name will return average results for these layers. Therefore, we need to take the invocation order into account so we can capture the information of each individual layer.

Table 2: Nvidia Tesla K40 and GTX1080 configuration details.

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

This imposes challenges to tie the characterized microarchitectural information to a specific layer. In order to address this challenge, we designed a database-backed trace tracking system that can capture the microarchitectural information for each layer in the CNN model. We establish a relational database to store platform-specific information, providing indices including the layer ID, kernel name, invocation order, and etc. With the help of this trace tracking system, we are able to obtain layer-specific micro-architectural information in a convenient and accurate manner. Figure 5 shows the overall workflow of the trace tracking system. We first profile the general execution information, i.e. kernel name and invocation order to create the relational database table, and then we leverage the database to profile the layer-specific microarchitectural metrics and extract the necessary information to drive our analytical tools.

3.4 Experimental Setup

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

4 EVALUATION RESULTS

In this section, we present several key metrics that capture performance in terms of microarchitectural details. Our evaluation is done on a kernel basis. Considering that cuDNN uses a flexible

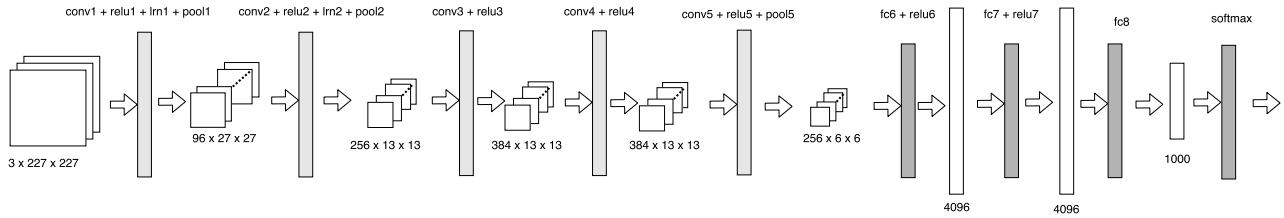


Figure 4: The organization of AlexNet.

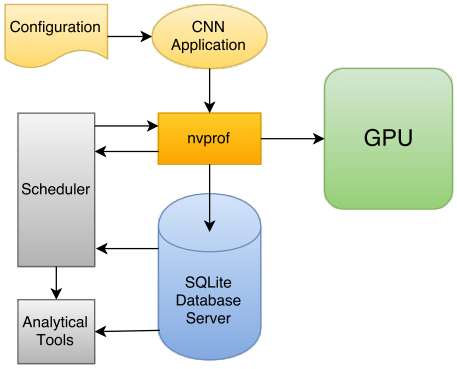


Figure 5: Database-based trace tracking system.

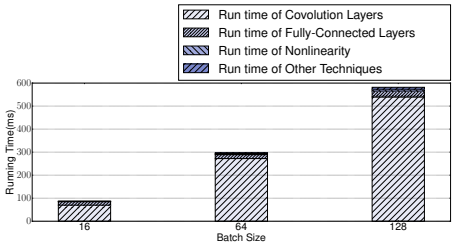


Figure 6: Runtime of AlexNet on the K40.

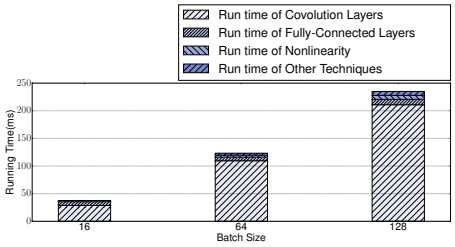


Figure 7: Runtime of AlexNet on the GTX1080.

strategy to instantiate kernels while varying template arguments for optimization purposes, we use the layer name rather than kernel name to present our evaluation results, even though the results are measured for kernels of primitives in cuDNN and cuBLAS.

4.1 Performance Analysis

First, we evaluate the runtime of each layer involved in one epoch of AlexNet, running across various batch sizes. This gives us a overview of performance for each layer, allowing us to identify the important steps during model execution. Since both the size of input image, as well as the training parameters, are fixed in the model, the batch size becomes the only variable that controls the scale of the final workload and size of the intermediate data. Figures 6 and 7 showcase the run times of AlexNet running on the K40 and GTX1080, respectively. During backward propagation, layers with trainable parameters should have at least two computations performing both data and backward propagation of weights. Bias is not considered in this paper, since the related computations are very simple.

From Figures 6 and 7, we can clearly see that the layers performing linear transformations are the major bottlenecks during the execution of the entire AlexNet model on both platforms. Convolution layers dominate performance of the linear transformations. This trend is consistent on both platforms, which shows that using a larger batch size leads to better throughput in terms of image processing, meaning that the both platforms achieve good scalability. The execution time is drastically reduced on the GTX1080, though the runtime of the other layers (other than the convolution layers) tends to become more prominent. Moreover, we noticed that the execution time of each layer is well-correlated with the number of operations indicated in Table 1. All of the linear data transformation layers take longer to finish.

In Figure 8 we report the speedup of running AlexNet on a GTX1080, using the K40 performance as a baseline. Generally, the GTX1080 has more SMs, (although there are fewer CUDA cores in each SM) and a higher clock rate in terms of both processing cores and memory, so the speedup is expected. But it can be observed that the performance gain for each layer varies. The convolution, fully-connected and pooling layers have significantly higher speedup than the activation (ReLU) and softmax layers. The LRN layer has relatively higher speedup during backward propagation versus forward propagation. Based on this observation, we find that the more advanced hardware has varied impact on the different layers while the floating-point instruction counts are basically equal in the applications built for each platform. The floating-point instruction counts of layers in backward propagation are listed in figure 9. Note that we only present the results from backward propagation because we noticed that the metric trends for the forward and backward propagation across every layer are very similar in most cases, and backward propagation is the most critical part during the CNN training. We present results for a batch size of 128, since

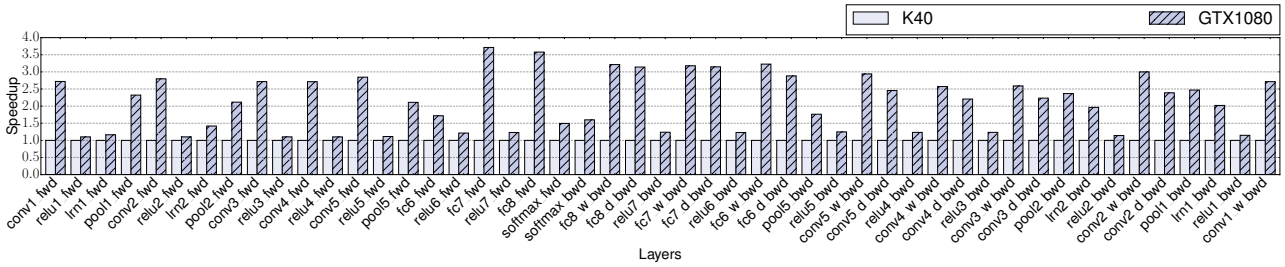


Figure 8: Speedup of running AlexNet on the GTX1080, using K40 performance as the baseline.

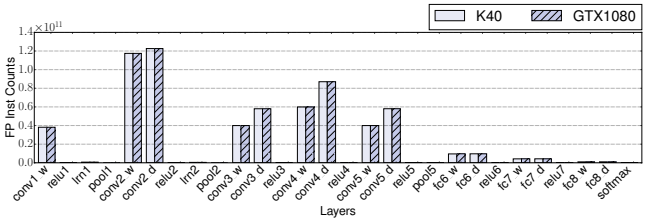


Figure 9: The floating-point instruction counts.

that this configuration can fully utilize the massive hardware of a GPU.

4.2 Characteristics Analysis of Layers

Next, we delve a step deeper into the microarchitectural details that can explain the difference in terms of performance observed across different layers. First, we highlight the reason for stalls during kernel execution on the baseline K40 platform in order to understand the characteristics of each layer. Figure 10 shows a breakdown of the stalls in each layer of AlexNet running on the K40.

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

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

In Figure 11, we present the stall breakdown when running AlexNet on the GTX1080. We can now see how the distribution of stalls change when running with a higher clock rate and with more SMs. One obvious finding is that the reason *stall_memory_throttle* is gone for all layers, meaning that the GTX1080 provides a faster data-path for moving data between processors and memory. For relu2, the major reason for stalls is tied to program characteristics that are highly dependent on memory operations. Increasing the memory frequency should lead to limited performance benefits, given that relu2 is a stream-like application with little temporal locality. The breakdown of stalls in conv2_w does not change much. Given that we see that the scheduler is choosing to run other warps, this layer still has headroom to improve if a higher GPU core clock rate is used, or SMs are added. The lrn2, pool2, and softmax layers become memory-bound on GTX1080, because the compute performance of the GTX1080 over the K40 has improved more versus the memory performance of the two systems. The fc6_w layer is both memory and compute bound. Since we see the same warp scheduling issues we encountered for conv2_w, there is headroom to improve performance for fc6_w. The new scheduler design in the Pascal architecture significantly alleviates the problems experienced with instruction fetching on the K40, so the new scheduler is able to handle more warps [17].

Finally, we list the CUDA cores ALU utilization for both platforms, considering that this metric reflects, to a great extent, how well the available hardware can be exploited. Figure 12 shows the ALU utilization while running AlexNet. As shown in the figure,

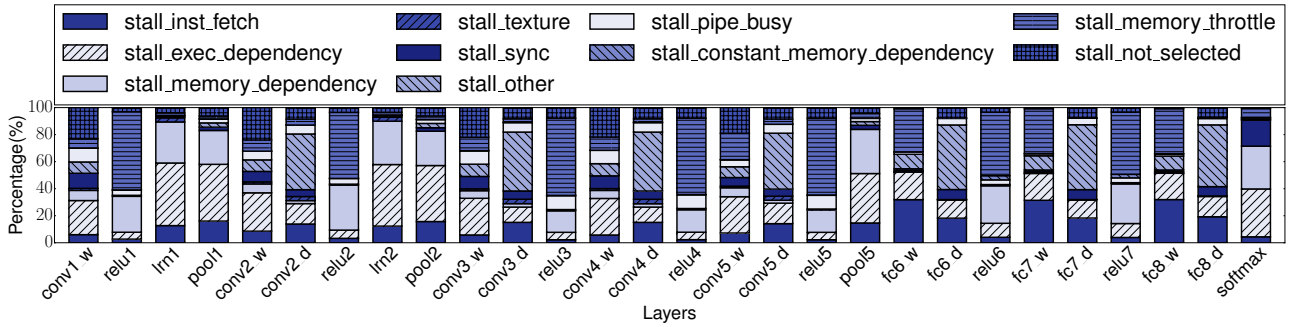


Figure 10: Stall breakdown for AlexNet running backward propagation on the K40.

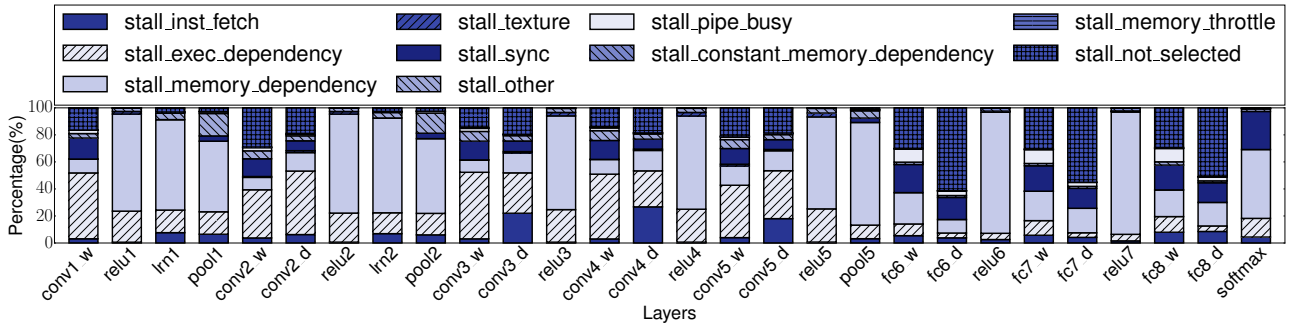


Figure 11: Stall breakdown for AlexNet, running backward propagation on the GTX1080.

almost all of the layers involved in linear transformations have higher ALU utilization levels on the GTX1080. There is one case in lrn1, where the utilization level on GTX1080 is lower. This is because the LRN layer becomes memory-bound on the GTX1080, due to the increased computing capability, requiring more data to be accessed. This results in more processing core idle time.

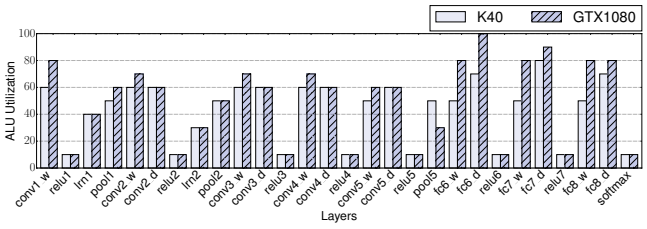


Figure 12: Compute unit utilization levels.

4.3 Memory Access Behavior

In this section, we focus on characterizing the memory behavior of each layer in AlexNet. Generally, in a discrete GPU, other than the main memory, the other critical memory component is the cache. The design of cache takes advantage of the locality present in applications, both in time and space, reducing the latency between instruction processing and memory access. In the GPU models we use, there are three different types of cache working together to support the streaming multiprocessors: i) an L2 cache, ii) a texture cache, and iii) an L1 cache, as shown in figure 3. Other than that, there is also an on-chip fast scratch-pad memory, shared memory, for the programmer to directly utilize in order to achieve better performance. Depending on the memory space specified by the CUDA programmer, the processor will initially request the data from either the L1, shared memory, or the texture cache. If not present in any of the three locations, the data will be requested from higher levels in the memory hierarchy.

Given that the K40 and GTX1080 have different on-chip memory arrangements, as observed in Table 2, we showcase the cache hit rate on both platforms. In this section, we only present a subset of layers in backward propagation, using a batch size of 128 for simplicity, because the layers of the same type have very similar characteristics, as discussed earlier in our stall analysis and in our utilization evaluation. Likewise, we select conv2_w/d, relu2, lrn2, pool2, fc6_w/d, and softmax to represent convolution, activation, LRN, pooling, fully-connected, and softmax layer, respectively. We also only present backward propagation with a batch size of 128 for the same reasons as in our earlier discussion. Figures 13 and 14 present the cache hit rates of all caching components.

From the cache hit rates shown for the K40 in Figure 13, we notice that layers of the linear data transformation make good use of the texture cache. This can be explained since the computations in both the convolution and fully-connected layers exhibit a high degree of spatial locality. The texture cache is designed in such a way as to take advantage of spatial locality. The L1 cache is a bit too limited in terms of space to handle this data. As a result, we see no L1 activities. In terms of the L2 hit rate, we can see cache accesses exist

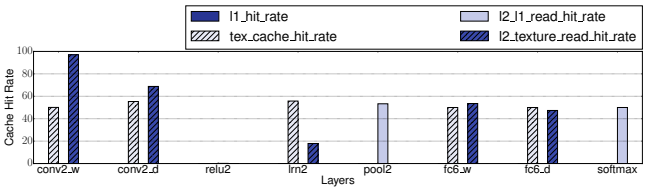


Figure 13: Cache hit rate for backward propagation of selected layers on the K40.

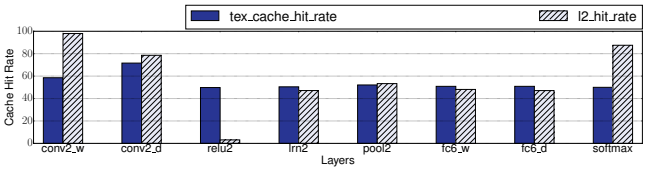


Figure 14: Cache hit rate for backward propagation of selected layers on the GTX1080.

in almost every layer except the activation layers. This is caused by the element-wise operations present, so the activation layer acts as a streaming application with no temporal locality. Likewise, the texture cache is heavily utilized as well on the GTX1080. Even the activation layer, which has no temporal locality, also makes use of the texture cache to exploit spatial locality. In order make a fair comparison, we enable the usage of L1 cache on the GTX1080 by toggling the corresponding compile flag [4]. However, the L1 cache is unified with the texture cache, so it is difficult to observe any L1 cache activities merely through the texture cache hit rate. To address this issue, we compare the texture cache hit rate between the two cases where L1 cache is enabled, and then disabled. The results show no change in the texture cache hit rate, meaning that there is basically no L1 activities as well.

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

Although there are many differences in memory performance between the two platforms, the trend in these metrics for the two platforms is still very similar in most cases. To provide a better view from both the processing core side and DRAM side, we split the metrics into two parts, each of which represents the memory components closer to the processor or closer to the DRAM, respectively. Figures 15- 16 shows the number of memory transaction and the memory throughput in various memory components. Note that we only present results from the GTX1080 because the trends are very similar.

From Figure 15a, we can see that the linear data transformation layers rely heavily on shared memory and the texture cache. As indicated in the cache hit rate figures, both the convolution and fully-connected layers possess high temporal and spatial locality, given that data accessed within a region is repeatedly accessed. As a result, there are a large number of memory transactions issued to these two memory levels, especially read requests. This means that some shared memory data is heavily reused during the computation. On the contrary, for other layers, the utilization of shared memory and texture cache is very limited. Even for the pooling and LRN layers, the data reuse rate is very low. Figure 15b supports the previous statement. For the other layers, including pooling, LRN, activation, and Softmax, the number of memory transactions does not vary significantly across the memory hierarchy.

With regards to memory throughput, In figure 16a, we can see that shared memory throughput was almost 4x higher than on the texture cache, even though the number of memory transactions in these two components is similar in the convolution layers. This indicates that shared memory has much higher bandwidth than the texture cache. For instance, shared memory usually takes 38 cycles to read, while the texture cache takes 436-443 cycles [29]. The latest hardware has shortened the performance gap between those two, but the gap is still significantly wide. Therefore, the bandwidth of texture cache is a limiting factor for the convolution layers.

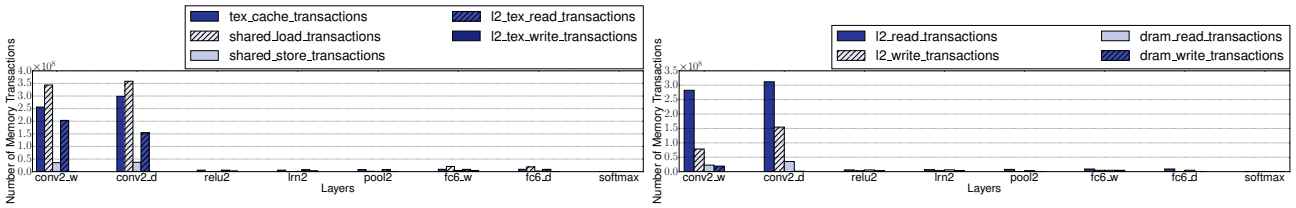
Note that increasing the bandwidth of texture memory without taking other associated memory components into account, could result in limited benefits. For example, if the texture cache becomes much faster than the L2 cache, execution will bottleneck at the L2. Increasing the bandwidth of the texture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

From Figure 16b, we can see that the throughput of DRAM in the activation layers is higher than that for the other layers. This is because the memory access pattern in the activation layers is more regular, meaning that memory requests can be coalesced, resulting in a better ratio between the size of useful data to number of memory transactions. Given that throughput is computed using the size of the requested data, divided by the time between the first and last memory transaction, a higher ratio leads to higher throughput.

We evaluate the utilization of the memory hierarchy in Figure 17. We show how each layer utilizes individual memory levels in the hierarchy. We find that the convolution layers can leverage shared memory and the texture cache, while activation layers utilize the DRAM heavily during the execution.

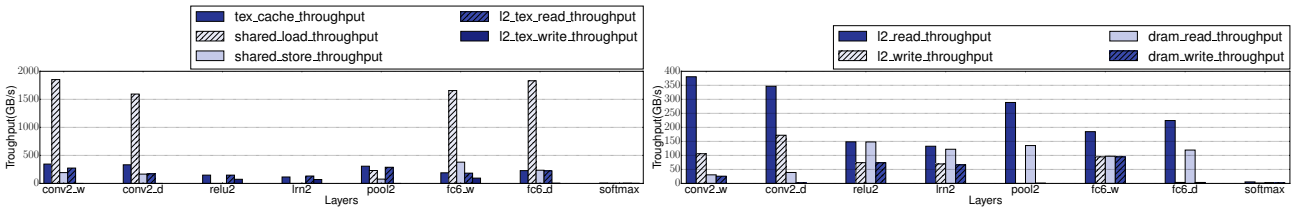
4.4 Potential Optimization

From our analysis, we propose a number of design changes that can benefit CNN execution on GPUs, especially the GTX1080. First, we begin with the major bottleneck which are present in the convolution layer, as indicated in figures 6 and 7. Stalls during convolution are due both to intrinsic program characteristics, and the limits of the hardware (even on the GTX1080). A simple solution is to add more SMs. Increasing the DRAM bandwidth on the GTX1080 will not benefit the CNN throughput very much. Instead, if we increase the bandwidth of the texture cache, we should see much better performance. As discussed earlier, a significant number of memory



(a) (b)

Figure 15: (a) Number of memory transactions in memory components closer to processor. (b) Number of memory transactions in memory components closer to DRAM.



(a) (b)

Figure 16: (a) Memory throughput of memory components closer to processor. (b) Memory throughput of memory components closer to DRAM.

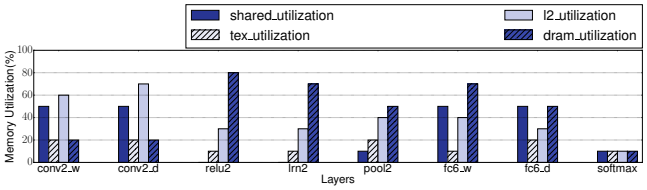


Figure 17: Memory components utilization of selected layers on the GTX1080.

transactions occur in the texture cache, but given its meager bandwidth, low throughput results. Thus, increasing the bandwidth of texture cache is beneficial in terms of reducing the read latency from the texture cache.

Next, we find that L1 cache is essentially unused in most of the layers. The main reason is that the L1 cache is too small to hold data that has a strided access pattern. Based on this observation, we can enable L1 cache bypassing[27] for selected layers to avoid unnecessary data requests to the L1 cache. When we re-run our application with the L1 cache disabled for both reads and writes, we observe a speedup in some layers for both forward and backward propagation. From these results, we find that a single layer used in backward propagation (calculating the convolution layer weights) can achieve a 6.2% speedup on the GTX1080. However, this approach is limited in terms of achieving better overall application throughput. One issue is that some layers exhibit temporal locality, so L1 cache bypassing needs to be applied selectively by these layers. If we focus on optimizations that only benefit a subset of layers, the overall performance gains will be limited.

Another optimization we explore is to apply kernel fusion[28] for the linear data transformation layers and the non-linear activation layers. As indicated in the results from the utilization breakdown, the activation layers place little pressure on compute resources due to its simple, element-wise, operations. The idea is to combine the linear data transformation and non-linearity. By doing so, we can eliminate all activation layers in the neural network model, leading to a significant reduction in the number of memory transactions, with only a small amount of computation added in the linear data transformation layers. Although the run time of the activation layers is insignificant, we can still save the time spent on running the driver and kernel launch, improving power efficiency as well. Given that cuDNN is not an open-source library, we are not able to further explore kernel fusion. So in order to evaluate the potential benefits of kernel fusion, we directly removed activation layers, assuming that the additional computation in the linear data transformation layers can be ignored. In this experiment, we measure the overall runtime, not just the kernel execution time. We are able to produce a speedup of 4% on average. As the activation layers only take 3% of the overall execution time, we save approximately 1% the time spent on driver and kernel launch.

5 DISCUSSION

Convolutional neural networks are quickly becoming very important applications in a number of domains. CNN computations have a set of applications with distinct characteristics both in computing and memory access. Given the diversity of CNN applications, exploring characteristics of the basic primitives in CNN is a prerequisite to accelerating this class of applications in general. Some researchers have explored using FPGAs for application-specific solutions [23]. However, the GPU is still preferred in most cases, given

that it provides a more flexible parallel programming framework and large memory space for storing massive amount of training data. Our evaluation in this paper focused on the microarchitectural demands associated with CNNs when mapped to two Nvidia GPUs. We evaluated how the same workload scales on platforms with different computing capabilities. We also analyzed microarchitectural metrics across different layers, considering the pressure placed on both compute units and memory components. We also proposed optimization methods based on the observed bottlenecks and insights. Through our experiments, we find there is still further room for improvements from perspective of both hardware and software optimizations.

As a further step, rather than reducing the execution time, we will focus on power efficiency. Considering that each layer has different needs in terms of computing resources and main memory, the GPU architecture can be argumented with big-little core techniques, so that heavy layers that hunger for compute resources can be scheduled on big cores, while lightweight layers can run on the smaller cores. Another approach is to design a dynamic clock rate tailored for each layer. It is demonstrated that not every layer requires the same processor and memory frequency, so finding a set of clock rate configurations for each layer can achieve better power efficiency.

6 RELATED WORK

There have been a number of earlier evaluation studies that focused on Neural Networks. Shi et al. conducted a series experiments of evaluating the current state-of-the-art deep learning software tools [24]. They evaluate a number of neural network models using state-of-the-art deep learning tools on both single and multiple GPUs. They propose a general guide of leveraging proper software tools on the targeted platforms, They also point out possible optimization directions for researchers. Kim et al. also evaluate several existing deep learning frameworks and suggest possible optimization methods leveraging convolution algorithms to improve CNN efficiency [10]. They characterize existing deep learning frameworks at an application level and explore the benefits of using different convolution algorithms in order to achieve better performance. Rhu et al. measure the memory usage of DNNs and propose a virtualization method to deal with issues of memory limits of a GPU [21]. Basically, they characterize the data access and data re-use patterns to create a virtual memory management strategy for DNN applications. The first two works above only evaluate performance at an application level, and the last focuses on data usage in the memory. The work presented in this paper provides a much more comprehensive dive into CNN execution behavior from a GPU microarchitectural perspective.

7 CONCLUSION

In this paper, we characterize the demands placed on a GPU microarchitecture while running a commonly used CNN model (AlexNet). We consider performance on a layer-by-layer basis. We carefully select metrics that can characterize the execution behavior of each layer in the model, and identify the major limiting factors for each layer. From our evaluation, we find that the characteristics of each layer vary significantly due to the distinct type of operations performed. Based on the microarchitectual demands imposed by each layer, we identify the major bottlenecks present in both an entire CNN model, and each individual layer, and suggest several optimization approaches that are able to improve the performance with only minor changes and overhead.

REFERENCES

[1] Martín Abadi et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). http://tensorflow.org/

[2] Tianqi Chen, Mu Li, and et al. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR* (2015). http://arxiv.org/abs/1512.01274

[3] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *arXiv preprint arXiv:1410.0759* (2014).

[4] NVIDIA Corporation. 2014. CUDA C Programming Guide. (2014).

[5] NVIDIA Corporation. 2015. CuBlas library v7.5. (2015).

[6] NVIDIA Corporation. 2017. CuDNN library v6.0. (2017).

[7] Shi Dong and David Kaeli. 2017. DNNMark: A Deep Neural Network Benchmark Suite for GPUs. *GPGPU-10* (2017), 63–72. https://doi.org/10.1145/3038228.3038239

[8] Erico Guizzo. 2016. How Google's Self-Driving Car Works. (2016).

[9] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093* (2014).

[10] Heehoon Kim, Hyoungwook Nam, Wookeun Jung, and Jaejin Lee. 2017. Performance Analysis of CNN Frameworks for GPUs. *Performance Analysis of Systems and Software (ISPASS)* (2017).

[11] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems 25* (2012), 1097–1105.

[12] Andrew Lavin and Scott Gray. 2015. Fast Algorithms for Convolutional Neural Networks. *arXiv preprint arXiv:1509.09308* (2015).

[13] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* (2015), 436âĂŞ444. https://doi.org/10.1038/nature14539

[14] NVIDIA. 2012. NVIDIA's Next Generation CUDA^{TM} Compute Architecture, Kepler^{TM} GK110. (2012).

[15] NVIDIA. 2016. CUDA Toolkit Documentation. (2016).

[16] NVIDIA. 2016. NVIDIA GeForce GTX 1080. (2016).

[17] NVIDIA. 2016. NVIDIA Tesla P100. (2016).

[18] NVIDIA. 2016. TESLA GPU ACCELERATORS FOR SERVERS. (2016).

[19] Genevieve B. Orr and Klaus-Robert Mueller (Eds.). 1998. *Neural Networks : Tricks of the Trade*. Lecture Notes in Computer Science, Vol. 1524. Springer.

[20] Victor Podlozhnyuk. 2007. FFT-based 2D convolution. (2007).

[21] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. *Microarchitecture (MICRO)* (2016).

[22] Frank Seide and Amit Agarwal. [n. d.]. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. ACM, 2135–2135. https://doi.org/10.1145/2939672.2945397

[23] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Es. 2016. From high-level deep neural models to FPGAs. *Microarchitecture (MICRO)* (2016).

[24] Shaohuai Shi, Qiang Wang, Pengfei Xu, and Xiaowen Chu. 2016. Benchmarking State-of-the-Art Deep Learning Software Tools. *CoRR* abs/1608.07249 (2016). http://arxiv.org/abs/1608.07249

[25] David Silver and Google DeepMind Demis Hassabis. 2016. AlphaGo: Mastering the ancient game of Go with Machine Learning. (2016).

[26] Theano Development Team. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints* abs/1605.02688 (May 2016).

[27] Yingying Tian, Sooraj Puthoor, Joseph L. Greathouse, Bradford M. Beckmann, and Daniel A. Jiménez. 2015. Adaptive GPU Cache Bypassing. *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*. https://doi.org/10.1145/2716282.2716283

[28] Guibin Wang, YiSong Lin, and Wei Yi. 2010. Kernel Fusion: An Effective Method for Better Power Efficiency on Multithreaded GPU. *Proceedings of the 2010 IEEE/ACM Int'L Conference on Green Computing and Communications & Int'L Conference on Cyber, Physical and Social Computing*. https://doi.org/10.1109/GreenCom-CPSCom.2010.102

[29] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. 2010. Demystifying GPU microarchitecture through microbenchmarking. *Performance Analysis of Systems and Software (ISPASS)* (2010).