The Dissertation Committee for Sangkug Lym
certifies that this is the approved version of the following dissertation:

Efficient Deep Neural Network Model Training by Reducing Memory and Compute Demands

Committee:

Mattan Erez, Supervisor

Andreas Gerstlauer

Michael Orshansky

Sujay Sanghavi

Jason Clemons

Efficient Deep Neural Network Model Training by Reducing Memory and Compute Demands

by

Sangkug Lym

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2019

Dedicated to my wife, Jung Min Lee

and my parents, Chun Geun Lim and Young Seok Choi.

Acknowledgments

First and foremost, I would like to thank my advisor Mattan Erez. Without his advice and guidance, I could have not accomplished this dissertation. He provided insightful critiques on my research work from high-level directions to technical details. He was always available and willing to join me in discussions to resolve my curiosity and problems. Importantly, I am very grateful to him for giving me great freedom in exploring and deciding research topics. This dissertation was impossible without his strong support for me to pursue what I desired to do and his willingness to expand his research expertise. Other than just research, I also learned many life lessons from him. I was privileged that I could pursue my Ph.D. with Mattan and no words can describe how much I appreciate him as an adviser.

I am grateful for the superb collaborators and mentors at NVIDIA: Donghyuk Lee, Micheal O'Connor, Niladrish Chatterjee, Jason Clemons, Burc Eryilmaz, Michael Andersch, and John Tran. I especially thank Donghyuk Lee for closely working with me. I appreciate their knowledge, experience, and advice. Thanks to their help, I could learn many important details on GPU architecture and industry-level deep learning performance optimization techniques. My experience at NVIDIA became the tipping point to research deep learning accelerators and led to this dissertation.

I also thank many outstanding collaborators and managers at Microsoft including Haishan Zhu, Ming Liu, Taesik Na, and Eric Chung. My internship at Microsoft was important in expanding my knowledge boundary associated with deep learning acceleration to FPGA and data center context. Also, technical discussions with my collaborators at Microsoft helped advance my thought process.

I also would like to show appreciation to professors and academic collaborators for their advice and feedback to improve my work. Especially, I thank Sujay Sanghavi for providing critical feedback on the algorithm side of my work and Songjun Lee at Hanyang University for encouraging me to start Ph.D. studies. I also thank my student collaborators: Wei Wen at Duke University and Heonjae Ha at Stanford University for their feedback that improves the quality of my research.

In addition, I am fortunate to know many talented graduate students during my Ph.D. I would like to give special thanks to Yongkee Kwon, Siavash Zangeneh, Benjamin Cho, Esha Choukse, Kyushick Lee, and Chun-Kai Chang for sharing their experience and motivating me to continuously improve my work through active discussions. I am also thankful to the senior members who encouraged me and provided me many research tips: Jungrae Kim, Don Wan Kim, Seong-Lyong Gong, and Wooseok Lee. I also thank other my Ph.D. friends at our research group: Song Zhang, Wenqi Yin, Tianhao Zheng, Anyesha Ghosh, Majid Jalili, Mike Sullivan. And, I am grateful for the friendship with many others: Alex Hsu, Ben Lin, Ali Fakhrzadehgan, Tian Tan, Jeageun

Jung, Mochamad Asri, Kishore Punniyamurthy, Kamyar Mirzazad Barijou, Zhuoran Zhao, and Shuang Song.

Lastly but not least, I would like to express the deepest gratitude to my loving family. I thank my parents Chungeun Lim and Youngseok Choi and my brother Sanghyun Lym for their strong support for my decision to pursue Ph.D. studies. Most importantly, I thank my wife Jung Min Lee for being always there with me. This dissertation would never have been made without her love and care. Lastly, I would like to say to her that we made it together.

Sangkug Lym

December 2019, Austin, TX

Efficient Deep Neural Network Model Training by Reducing Memory and Compute Demands

Publication No. ____________________

Sangkug Lym, Ph.D.
The University of Texas at Austin, 2019

Supervisor: Mattan Erez

Deep neural network models are commonly used in various real-life applications due to their high prediction accuracy for different tasks. In particular, CNN (convolutional neural network) models have become the de facto choices for most vision applications such as image classification, object segmentation, and object detection. Modern CNN models contain hundreds of million of parameters and training them requires millions of computation- and memory access-heavy iterations. To reduce this expensive CNN model training cost, this dissertation presents computation and memory cost-efficient training mechanisms with a combination of workload scheduling, learning algorithm, and accelerator architecture optimizations. This dissertation also introduces a performance model for data-parallel accelerators as a fast and accurate method to estimate the performance impact of the proposed architectural optimizations and to help fine-grain accelerator design space exploration.

The first part of this dissertation discusses reducing the memory bandwidth demand for CNN training. I first analyze data reuse opportunities in CNN training and show that CNN training has high data locality between network layers but that conventional training mechanisms fail to utilize this inter-layer locality. Then, I develop a CNN training scheduling mechanism that modifies the network execution ordering in a way that captures the inter-layer locality while supporting high compute resource utilization. I also introduce a training accelerator that adopts architectural optimizations that hide additional data transfers caused by the proposed scheduling modification and realize effective training speedup. The proposed training accelerator has 45 mixed precision FLOPS and, with the memory bandwidth-efficient network training scheduling, beats a state-of-the-art GPU that has $\sim$3X higher peak FLOPs.

The second part of this dissertation focuses on reducing the computation cost of CNN training. To reduce computations during training, I use neural network model pruning from the beginning of training. The insight is that a fully trained CNN model contains many non-critical parameters and pruning such parameters during training has only a minor impact on the learning quality. I also choose to structurally prune these parameters to provide high data parallelism avoiding complex data indexing, thus maintaining high compute resource utilization. For the practical implementation of pruning while training, I propose three algorithmic optimizations. Theses optimizations are designed to remove the need for the memory accesses caused by tensor reshaping, reduce the number of training runs in finding the desired pruning hyper-parameters, and maintain high data parallelism even for processing a highly pruned CNN model. Overall, the proposed algorithm speeds up the training of commonly used state-of-the-art image classifiers by 39% with only 1.9% accuracy loss.

The third part of this dissertation deals with training pruned CNN models on accelerators with large systolic arrays. I first show my observation that processing structurally-pruned CNN models on a large systolic array severely underutilizes its PEs (processing elements) because the reduced number of channels decreases parallelism. Then, I show that naively splitting a large core into multiple small cores improves PE utilization but decreases input reuse and incurs >4% area overhead. To improve PE utilization and maintain high input reuse, I propose a flexible systolic array architecture that can reconfigure its structure to one of several modes, each designed for efficient execution of CNN layers with different dimensions. I also develop compile-time heuristics that optimize mapping the layer workload to the flexible systolic array resources for both high performance and energy efficiency. My new mechanisms increase PE utilization by 36% compared to a single large-core design and improve training energy efficiency by 18% compared to many-small-core designs.

The last part of this dissertation is about developing an accelerator performance model for accurate CNN execution time estimation. For accurate performance modeling, I introduce a memory traffic model that predicts the data traffic at different levels of the GPU memory system hierarchy. This involves an in-depth analysis of the memory access patterns of data-parallel convolution kernels and the spatial locality. I demonstrate that the proposed performance model can provide guideline to fine-tune the GPU resources for efficient CNN performance scaling.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Training modern DNNs (deep neural networks) requires millions of computation and memory bandwidth intensive workload iterations. In addition, ever-growing network architecture complexity and training dataset size have been continuously increasing the already expensive DNN model training cost [11]. Even using a cluster of training accelerators with large compute throughput, training a state-of-the-art network model on a large dataset such as ImageNet [22] still takes multiple hours to days [29]. Furthermore, unlike inference, training workloads involve many memory bandwidth-bound vector functions such as feature normalization. This increases the need for high memory bandwidth leading to many modern training accelerators adopting expensive high-bandwidth memory system. This dissertation focuses accelerating convolutional neural network training, an important branch of DNN models for computer vision tasks, by identifying and reducing its unnecessary memory accesses and unimportant computations.

1.1 High Memory Bandwidth Needs in CNN Training

Convolutional neural networks (CNNs) are the state of the art for various vision applications [57, 85, 84, 101]. Modern CNN models have millions of learning parameters and they are first trained on a large dataset to be used for inference. CNN architectures consists of different layer components and they need high memory bandwidth to fully utilize the computing elements on high performance accelerators. However, based on my observation, commonly used data-parallel CNN training mechanisms on current systems require 3–4 times more memory bandwidth than necessary, reducing performance and wasting energy.

Conventional CNN training propagates a mini-batch of samples across network layers in lockstep, where a mini-batch typically consists of 32–512 samples [96, 95, 36]. Therefore, the mini-batch output of one layer is reused in its following layer as an input. However, large mini-batches have per-layer memory footprints that exceed typical on-chip buffer capacity. This leads not capturing the inter-layer data locality, resulting in high memory traffic. Many inter-layer locality techniques were proposed in the context of network inference [77, 5]. However, directly applying these techniques to training is ineffective or not feasible because they do not optimize locality across large mini-batches, and their design is not compatible with batch normalization [46], the most commonly used feature normalization algorithm. In this dissertation, I introduce a memory bandwidth-efficient training mechanism that enables inter-layer data reuse and a coupled training accelerator that achieves effective training speedup.

1.2 Model Pruning for Cost-efficient CNN Training

CNN architectures contain millions of learning parameters (or weights) for learning complex features. However, not all parameters equally contribute to prediction accuracy. Model pruning is a commonly used technique that removes less critical learning parameters from an initially-dense model [33, 34]. This leads to lower memory and compute cost with minor prediction accuracy loss compared to the original dense model.

Model pruning mechanisms are typically used for high-performance and energy-efficient inference [103, 25, 39, 38]. They typically retrain a pre-trained model, which eventually increases net training time. Some prior work prunes models during training [110, 103, 102, 113, 4]. However, this work maintains the original network architecture without pruning, or prune only a limited number of times during training. In this dissertation, I discuss a CNN training mechanism that constantly prunes unimportant parameters during training to gradually reduce computation cost over training. Furthermore, I develop and evaluate a practical implementation of this algorithm for modern CNN models.

1.3 Efficient Pruned Model Training on High-Throughput Accelerators

Not all parameters of a CNN model are equally important. Structured model pruning removes such parameters at the granularity of channels. Therefore, unlike unpruned CNN models that have regular number of channels such as 64, 128, and 256 (powers of two) [57, 85, 84, 101], their channel-pruned versions have an arbitrary number of channels, such as 3 and 71. Such convolution layers are computed as GEMMs (general matrix multiplies) for efficient execution on data-parallel accelerators. However, the GEMMs of the pruned layers have arbitrarily reduced dimensions and these GEMMs execute inefficiently on the large GEMM cores used in modern training accelerators [21, 86]; many GEMM tiles of pruned CNNs tend to have small sizes and executing them on a large core does not fully utilize its PEs.

Splitting a large core into many small cores can improve PE utilization. This is because GEMM is tiled into smaller parts and these small tiles are processed by small cores with high PE occupation (mitigating internal PE utilization fragmentation). However, this core splitting incurs >4% area overhead for data paths and splitting SRAM buffers. Also, such a many-core design increases on-chip input traffic for fetching duplicated inputs to multiple cores, hurting the energy efficiency of training and requiring high on-chip BW. In this dissertation, I introduce a flexible GEMM core architecture that can reconfigure its structures to achieve both high PE utilization and high input reuse. I also discuss GEMM tiling and scheduling techniques for efficient utilization of the reconfigurable hardware resources.

1.4 The Need for an Accurate Performance Model

GPUs are the most widely used training accelerators. The increasing demand for high computation throughput of DNN training is driving GPU arithmetic performance, which has been growing at higher than its historical rate [76]. Based on my analysis, compared to the rapid GPU compute throughput increase of 32× over the past 9 years, GPU memory system bandwidth has improved by only 13×. This memory wall problem can bottleneck the performance of even the arithmetically-intensive CNNs, making performance scaling challenging.

It is therefore imperative to balance both arithmetic and memory performance in architecting a future GPU or other accelerators for efficient CNN performance scaling. This balanced designing benefits from analytical modeling, which can quickly provide insight and narrow the design space before slower and more resource-consuming modeling is used (e.g., simulators [1, 80]). Analytical models also aide in the optimization of software for efficient hardware resource utilization [114, 61]. Given that DNN models evolve and change quickly, the benefits of a performance model are significant. In this dissertation, I present an accurate GPU performance model that relies on an accurate memory traffic modeling. I also discuss using the performance model for fine-grain GPU resource allocation for cost-efficient CNN performance scaling.

1.5 Thesis Statement

Ever increasing neural network model complexity and training dataset size have been constantly raising CNN training cost. Partially serializing a mini-batch CNN training workload enables inter-layer data reuse in training and significantly reduces redundant memory traffic. Also, structurally pruning unimportant model parameters during training effectively reduces total training compute operations and leads to faster model training. At the same time, given the changes in the training workload scheduling and algorithm, co-optimizing the accelerator architecture is critical to maintaining high compute resource utilization and significantly improves the training performance of modern CNNs.

1.6 Contributions

In this dissertation, I resolve and mitigate the two key overheads of CNN training; high memory bandwidth requirement and high computation cost. I first resolve the memory bandwidth problem of training by: (1) partially serializing a mini-batched training workload in a way that utilizes inter-layer data locality using the available on-chip storage resources, and (2) co-optimizing the training accelerator architecture to cope with the challenges from the changes in the scheduling and algorithm. Next, I reduce the cost of CNN training by gradually pruning less critical parameters from the initial dense model. I introduce three algorithm techniques for efficient model pruning that maintains high training accelerator resource utilization. I also co-design a training accelerator to resolve the reduced PE utilization in processing pruned models on a large systolic array by making the core structure change depending on GEMM dimensions. To evaluate the proposed ideas, I devise a training accelerator performance model that accurately estimates the memory traffic and the performance of CNNs. I also explore the architectural optimization space of a GPU, the most common DNN training accelerator using my model. I summarize the contributions of this dissertation:

1. I propose Mini-Batch Serialization (MBS), a memory bandwidth-efficient network training mechanism. MBS decreases the memory traffic of modern CNN training by exploiting data reuse between layers. MBS breaks a mini-batch into sub-batches, which are then processed serially such that the inter-layer data of an entire sub-batch fits in an accelerator's on-chip buffers and can be reused. Implementing MBS raises two challenges. First, mini-batch wise feature normalization commonly used in modern CNNs is not compatible with (partially) serializing a mini-batch. Second, the data parallelism per sub-batch is smaller than for a whole mini-batch and can decrease the compute array utilization and increase core idle time. I resolve these challenges by (1) adapting an appropriate feature normalization algorithm to the context of MBS, (2) devising a sub-batching mechanism that adjusts the sub-batch size for the network layers to reuse inter-layer data with only a small impact to data parallelism, and (3) using a systolic data flow along with architecture optimizations that maintain high computing array utilization. Compared

to the conventional mini-batch wise training for modern CNNs, MBS with the proposed training accelerator optimizations reduces memory traffic by 4X and improves performance by more than 50%.

2. I propose PruneTrain, a CNN training acceleration mechanism that continuously prunes the model during training from scratch. For efficient execution on data-parallel training accelerators, I group parameters at channel granularity and prune those channels for which all parameters are below a threshold using group lasso regularization. This periodic reconfiguration maintains a still dense, yet smaller model. This model, which requires less computation, memory, and communication, continues to shrink as pruning continues throughout training. For performance-efficient pruning, I introduce three key optimization techniques. I first propose a systematic method to set the group lasso regularization penalty coefficient that controls group lasso regularization strength and achieves a high model pruning rate with small impact on accuracy with a few training runs. Second, I propose channel union, a memory-access cost-efficient and index-free channel pruning algorithm for modern CNNs with short-cut connections. Lastly, I propose dynamic mini-batch adjustment that dynamically increases the size of the mini-batch to fully utilize the available off-chip memory space for training iteration. PruneTrain speeds up the training of the commonly used state-of-the-art image classifier by 39%.

3. I propose FlexSA, a flexible systolic array architecture that improves PE

utilization while keeping input reuse high when processing pruned models. FlexSA is comprised of four systolic array cores. However, unlike a naive four-core design, where each core operates independently, FlexSA supports three additional inter-core operating modes, which achieves low on-chip traffic via inter-core data reuse. To efficiently utilize the underlying resources of FlexSA, I propose compile-time GEMM tiling heuristics that use large GEMM tiles with inter-core operations in 96% of cases, improving on-chip input reuse by 1.8× compared to a baseline four-core design. Overall, compared to the baseline four-core design, FlexSA adds only 1% additional area but improves energy efficiency by 18%. Also, compared to a single large systolic array, FlexSA improves average PE utilization by 1.4×.

4. To explore the design space of a GPU, the most common accelerator for DNN model training, I propose DeLTA, a GPU performance model based on accurate data traffic estimation at all levels of GPU memory system hierarchy. DeLTA separately models the traffic at each memory level using the access patterns of the data-parallel convolution algorithm. DeLTA also accounts for how the computation is blocked for locality and parallelism and how the hardware handles memory accesses in the caches and the software-managed shared memory. The estimated memory traffic is used to predict convolution layer execution time under different system configurations and identify the GPU performance bottlenecks. Overall, DeLTA predicts all L1, L2, and DRAM traffics

by GMAE less than 5% (geometric mean absolute error) and execution time by 6.0% GMAE for NVIDIA TITAN Xp GPU. I also show that DeLTA helps fine-grain GPU compute and memory resource balancing for cost-efficient CNN performance scaling.

1.7 Dissertation Organization

The remainder of this dissertation is organized as follows: Chapter 2 reviews the background for understanding CNN architecture, its model training mechanisms, a data-parallel convolution kernel, and training accelerator architectures. Chapter 3 proposes a memory bandwidth-efficient CNN training mechanism with the combination of workload scheduling, learning algorithm, and architecture optimizations. Chapter 4 discusses computation-efficient CNN training using an efficient model parameter pruning mechanism during training. Chapter 6 introduces the models to estimate memory traffic and performance of GPU for CNNs. Finally, Chapter 7 concludes the dissertation.

Chapter 2

Background

This chapter provides background on CNN applications and training accelerator architecture. I first explain different layer functions of CNNs and CNN training algorithms in Section 2.1. Then, I briefly explain the neural network model pruning mechanisms in Section 2.2. Lastly, I describe the architectures of commonly used CNN training accelerators and the data-parallel convolution layer kernel design for GPUs in Section 2.3 and Section 2.4, respectively.

2.1 CNN Architecture and Training

CNN Architecture. CNNs consist of various types of layers (Figure 2.1): convolution layers extract feature maps (or features or activation) from input images, down-sampling (e.g., pooling) layers reduce the size of each feature map, and fully-connected (FC) layers generate prediction scores for classes. Also, a convolution layer typically includes feature normalization and activation. Feature normalization maintains stable feature distributions across layers and different mini-batches (a set of samples used per training iteration) [46] and activation layers, e.g., ReLU [73] add non-linearity to the learning function.

Each convolution layer takes IFmaps (input feature maps) and produces OFmaps (output feature maps). As shown in Figure 2.1, each set of IFmaps and OFmaps has four dimensions: the number of samples in a mini-batch (B), the number of channels (C), and the height (H) and width (W) of each feature map. I use i and o to denote IFmap and OFmap, respectively.

Convolution and FC layers are implemented primarily with matrix multiplication and accumulation. As illustrated in Figure 2.1, a convolution layer convolves its IFmaps with small filter matrices (each $H_f \times W_f$), possibly with a non-unit stride. Filters hold weights and there are $C_i \times C_o$ filter matrices that map input to output channels. After each convolution, the results from all input channels are accumulated to a single output data point. As the same filter is reused for convolving IFmaps and the same input feature data is reused to compute different OFmaps, a convolution layer typically exhibits large data reuse and its performance is bottlenecked by compute through-

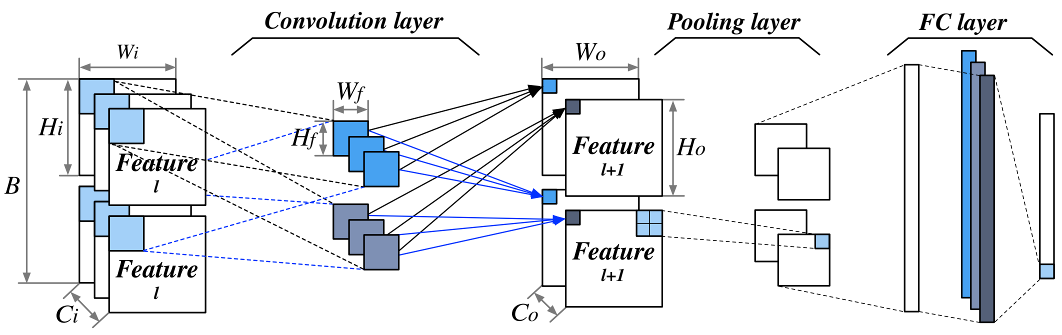


Figure 2.1: Layers of a CNN, and the feature and filter data structure of a convolution layer.

put [15]. Other layers, including pooling, activation, and feature normalization have low arithmetic intensity as their major function is element-wise vector computations [57, 46]. Therefore, their performance is constrained by data access bandwidth.

CNN Training. A modern CNN model contains millions of weights $\boldsymbol{W}$ (or learning parameters) that are trained using a collection of samples called the training dataset. The baseline training approach starts by randomly taking a sample from the training set and propagating it through the CNN layers to predict an output using the current weights. Next, the predicted output are compared to the ground truth, and loss l (or prediction error) is computed. Finally, the loss is back-propagated through the network layers, calculating the gradient of loss w.r.t. the current weights in each layer and then updating the current weights using the gradients.

Mini-batch SGD (stochastic gradient descent) is the most commonly used CNN training algorithm, which uses a set of training samples per training iteration for network propagation, loss computation, and model update. Using a large mini-batch exhibits many benefits: (1) it provides abundant data parallelism to each layer operation, which helps achieve high HW resource utilization, (2) it reduces the frequency of weight updates, and (3) it decreases the variance in weight updates between training iterations [63, 29, 20]. Typically, a mini-batch is 32–512 samples (possibly distributed across multiple processors) [94] but even a huge mini-batch bigger than 8K is often used with algorithmic support to enable extreme-scale distributed training [108, 2, 48].

Equation 4.1 describes the mini-batch SGD algorithm. Here f is the network's prediction on the input x_i, $\boldsymbol{W}$ are the weights, l is the classification loss function between the prediction and its ground truth y_i, and N is the mini-batch size.

$$\min_{\boldsymbol{W}} \left(\frac{1}{N} \sum_{i=1}^{N} l(y_i, f(x_i, \boldsymbol{W})) \right) \tag{2.1}$$

Feature Normalization for Fast Model Convergence. Feature normalization is critical to scaling the depth of a network architecture and enabling faster convergence of the SGD training algorithm. It maintains a constant feature distribution for the input to each convolution layer and avoids the parameter variance caused by random sampling. BN (batch normalization) is the default technique for most modern deep CNNs. BN normalizes features each mini-batch [46]. BN typically uses 32 or 64 samples (partial mini-batch assigned to a processor) for normalization. The performance of BN is highly memory bound because each of the mean, variance, and normalization steps requires reading of the entire mini-batch [52].

Distributed Training. A cluster of GPUs is typically used to train a complex CNN model on a large dataset. Data parallelism is the most commonly used multi-processor training mechanism [57]. First, each GPU in the system holds the same copy of the weights. Then, a mini-batch of input samples is distributed to each GPU and all GPUs process the inputs in parallel. Data parallelism is network-traffic efficient as the inter-GPU communication is required only for model updates; the partial weight gradients of all GPUs are first reduced then used to update the current weights. Although using more GPUs increases the peak computation throughput, it also increases this communication overhead, preventing linear end-to-end training performance scaling. For efficient weight gradient reduction, a ring-allreduce based communication is for weight gradients reduction, which efficiently pipelines data transfer latencies among nodes [104]. In particular, recently proposed hierarchical allreduce communication [64] reduces the communication complexity by hierarchically dividing the reduction granularity and achieves more linear training performance scaling with increasing number of GPUs.

Training Memory Context. Processing a training iteration requires a large off-chip memory space. This is mainly because the input activations of each layer at forward propagation should be kept in memory and reused to compute the local gradients in back-propagation. In particular, the total size of all layer inputs linearly increases with mini-batch size [69]. Therefore, small off-chip memory capacity or a large feature size of a CNN can constrain the mini-batch size per accelerator, and hence also the data parallelism of each layer. This eventually decreases HW resource utilization. In addition, insufficient memory increases the total number of training iterations per epoch because of smaller mini-batches, which increases the communication cost for model updates.

Chen et.al propose a memory-storage efficient training technique that stores the activations of only a fraction of layers [13]. Their approach keeps only the layer output activations that are expensive to compute and recompute the outputs of other layers during back-propagation. Although this method reduces the memory cost from $O(n)$ to $O(\sqrt{n})$, it increases computation during back-propagation.

2.2 Neural Network Model Pruning for Fast Inference

Model pruning has been studied primarily for CNNs, to make their models more compact and their inference fast and energy-efficient. Most pruning methods compress a CNN model by removing small-valued weights with a fine-tuning process to minimize prediction loss [33, 34]. Pruning algorithms can be unstructured or structured. Unstructured pruning can maximize model-size reduction but requires fine-grained indexing with irregular data access patterns. Such accesses and extra index operations lead to poor performance on deep learning accelerators with vector or matrix compute units despite reducing the number of weights and FLOPs (floating point operations) [31, 109, 8]. Structured-pruning algorithms remove or reduce fine-grained indexing and better match the needs of hardware and thus effectively realize performance gains.

2.2.1 Trial-and-Error Based Structured Model Pruning

One approach to structured pruning is to start with a pre-trained dense model and then attempt to remove weights in a structured manner, generally removing channels rather than individual weights [39, 43, 72, 38]. Unimportant channels are removed based on the value of their weights or hints derived from regression [97]. The removed channels are rolled back if accuracy is severely affected. Although effective, the search space of such a trial-and-error based model pruning substantially increases with the complexity of the network model, which can increase pruning time significantly. Also, as pruning is applied to a pre-trained model, these mechanisms do not speed up training.

2.2.2 Model Pruning with Parameter Regularization

An alternative mechanism to trial-and-error pruning uses *parameter regularization*. This optimizes training loss while simultaneously forcing the absolute values of weights or groups of weights toward zero. I call this process of forcing weights toward zero *sparsificiation*. Group lasso regularization is typically used to structurally sparsify weights by assigning a regularization penalty to l_2-*norms* of groups of weights [103, 102, 25, 4, 113]. This regularization-based pruning mechanism adds regularization loss terms to the baseline classification loss function as shown in Equation 4.1, then back-propagate the loss to update the weights to both improve accuracy and reduce their absolute values. Eventually, the sparsified weights can be effectively zeroed-out and pruned from the model.

2.3 Training Accelerator Architecture

2.3.1 GPU Architecture

GPUs are designed to accelerate compute-intensive highly-parallel workloads thus they require applications to express parallelism with many threads for efficient utilization. GPUs contain a large number of very wide SIMD (single instruction multiple data) cores, called streaming multiprocessors (SMs). In NVIDIA GPUs, each SM processes a warp of 32 threads in lockstep, such that ideally all 32 threads execute the same instruction. In addition to the processing elements, each SM contains load store units (LSU), register files (RF), a shared RAM (SMEM), and an L1 cache. The SMs share access to the L2 cache and DRAM through a crossbar interconnection network.

GPU workloads are tiled into thread groups called cooperative thread arrays (CTAs). The CTA scheduling mechanism is assumed to assign SMs to CTAs in a round-robin manner [62]. Each CTA typically consists of multiple thread warps that execute concurrently to hide memory access latencies. Given sufficient resources (RF and SMEM), multiple CTAs can be simultaneously executed within one SM (active CTAs). Interleaving multiple CTAs improves the ability of the SM to overlap computation with memory access.

2.3.2 Systolic-Arrays

A systolic array used for DNN training acceleration is (typically) a two-dimensional mesh of many simple and efficient processing elements (PEs). At each cycle of a kernel, each PE applies the same computation to its inputs and then passes the computed result or its unmodified inputs to one or more of its neighbors. All PEs communicate only with adjacent PEs such that there is minimal data movement and high computational concurrency [58]. Computation consists of pipelining inputs from the top and left (for example) edges of the array and obtaining results at each PE or at the bottom depending on the systolic dataflow. The large compute throughput required for convolutional and fully-connected layers, along with the repetitive computation and large data reuse are a good match for a systolic array, as found in Google's TPU ML accelerators [51, 21]. I describe the detailed systolic array operation in Section 3.3.1 along with the accelerator architecture that this dissertation proposes.

2.4 Data-parallel Convolution GEMM

Im2col (image-to-column) is one of the most commonly used algorithms for convolution kernels for data-parallel accelerators [19, 78, 42]. Im2col transforms the direct convolution described in Figure 2.1 into a single general matrix-matrix multiplication (GEMM) with three-dimensions ($M \times N \times K$) by merging the IFmaps and small filter matrices (Figure 2.2). In this example,

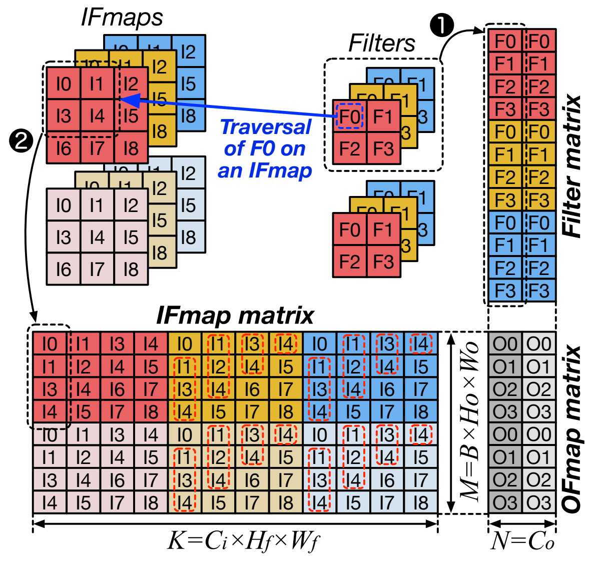


Figure 2.2: Im2col conversion: direction convolution converted to GEMM.

first, the 2×2 filters for each OFmap channel are converted into columns and stacked ❶. Next, the data elements in the IFmaps are laid out in a way such that the elements to be multiplied by one filter (F0) are placed as a column ❷. The IFmap matrix data layout changes depending on the filter size and stride. Im2col duplicates many data elements (data in red doted boxes) which represents input feature data reuse. Therefore, compared to the typical GEMM, convolution GEMM has greater data reuse and benefits more from caches or programmable buffers.

Chapter 3

CNN Training with Inter-layer Data Reuse

This chapter covers my memory bandwidth-efficient CNN training approach that reuses inter-layer data. I first discuss the inter-layer data reuse opportunity in CNN training and the difficulty of capturing such locality when using conventional training mechanism. Then, I introduce a CNN training approach that reuses inter-layer data with a small on-chip storage budget. The proposed training method changes the scheduling of CNN training along with adopting a coupled feature normalization algorithm. This chapter also introduces a systolic array-based training accelerator that supports high PE utilization given the proposed approaches. [1]

3.1 Data Reuse in CNN Training

CNN training consists of forward and back-propagation phases as described in Section 2.1. Figure 3.1 illustrates the major data elements needed for training and their reuse patterns with red arrows indicating opportunities for on-chip buffers to reduce memory bandwidth requirements and black arrows

[1]This chapter was published in the proceedings of The Conference on Systems and Machine Learning (SysML) in April 2019. Sangkug Lym contributed to this work as the main author.

indicating accesses to main memory. In both phases there is direct producer-consumer locality between layers—*inter-layer data* that can be buffered if it is not too large. The outputs of convolution, normalization, and activation layers in forward propagation (x, y, and z in the figure) are immediately used by their following layers. Normalization layers exhibit additional reuse because they iterate over inputs to first compute the mean and variance before normalizing the data [46]. The convolution outputs and the activations are stored in off-chip memory for reuse in back-propagation because their large storage requirements and long data reuse distance prevent on-chip buffering.

Back propagation exhibits even greater potential for inter-layer reuse. The loss gradients (w.r.t. x) are reused twice by a convolution layer to compute the gradients of weights and loss (w.r.t. z). Also, the convolution output stored in memory is reused multiple times to compute the gradients of the normalization layer parameters and the loss gradients (w.r.t. x). Activations

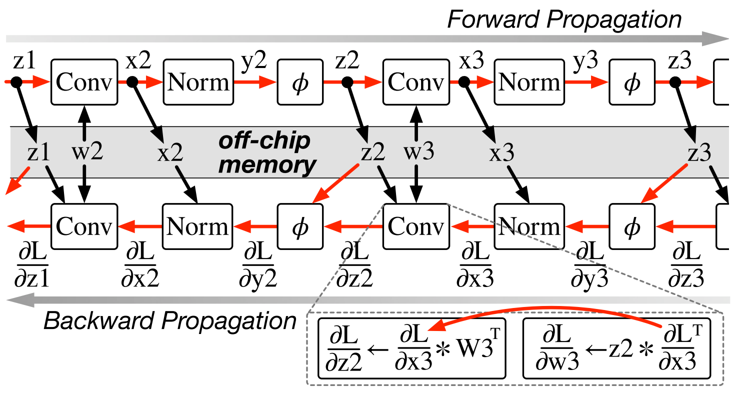


Figure 3.1: Dataflow in forward and backward propagations. Red arrows show the reusable data between layers.

read from memory are also used twice: z is used for convolution gradients and the derivative of z for activation gradients.

Multi-branch Network Modules. Recent CNNs typically consist of multi-branch modules that enable a deeper network architecture and more accurate prediction [96, 95, 36]. The residual [37] and inception [96] modules are the most representative multi-branch modules. A residual module has two branches whose outputs are accumulated when they merge while an inception module has numerous branches whose outputs are concatenated at a common node. As the input at the branch divergence is used by all branches and the outputs at the join can be referred to other branches for reduction, the multi-branch modules exhibit data locality between branches.

The Problem with CNN Training Memory Footprint. Mini-batch SGD increases each layer's memory footprint, and this limits the opportunity to reuse data on chip. Figure 3.2 shows the per-layer footprint of ResNet50 [36], the most common modern CNN, with a mini-batch size of 32 and a word size

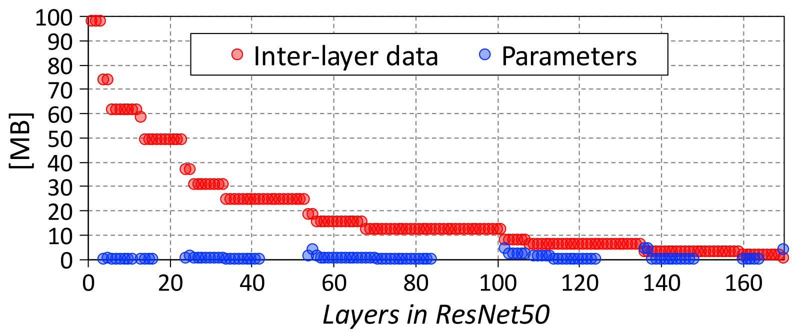


Figure 3.2: The size of inter-layer data and parameters of each layer in ResNet50 (sorted by inter-layer data size).

of 16b in the forward phase. Only 9.3% of inter-layer data can be reused even with 10MiB on-chip storage, leading to significant memory bandwidth waste for storing and refetching data. This problem is even more severe for larger mini-batch sizes, which are desirable as per-processor arithmetic performance and main memory capacity improve.

3.2 Memory BW-Efficient CNN Training with Mini-Batch Serialization

3.2.1 Serialization Principle

The primary goal of MBS (mini-batch serialization) is to improve reuse by exploiting inter-layer data locality. The key to MBS is partially serializing a mini-batch (propagating a small sub-set of a mini-batch at a time) to control per-layer memory footprint without impacting training accuracy. MBS is based on my insight that if the data synchronization points for functional correctness are maintained and an appropriate normalization algorithm is adapted, even processing a single sample at a time through all network layers does not alter the training result. The trivial serialization of one sample at a time, however, has two crucial drawbacks.

First, while baseline training reads weights and writes weight gradients just once per layer, full serialization re-reads weights and their partial gradient sums for each sample and updates the partial sums once per sample as well. Second, data parallelism within a single sample can be limited in some layers, degrading resource utilization and performance (especially when mapping to a highly-efficient systolic architecture). It is possible to process multiple samples at a time (a *sub-batch*) to to provide some intra-layer weight reuse and extra parallelism. However, the footprints of early layers are large and only a small sub-batch can be formed (1–2 samples), limiting the benefits of this approach.

MBS goes much further and balances locality of intra-layer weight reuse and parallelism with inter-layer locality. I do this by varying the number of samples per sub-batch across layers such that layers that can support more samples require fewer iterations and can benefit from the greater parallelism and locality. This is possible because down-sampling layers decrease feature map size and volume for deeper layers.

3.2.2 Layer Grouping Optimizes Reuse

Optimizing layer groups balances intra- and inter-layer locality trade-offs. The MBS algorithm forms initial layer groups by grouping adjacent layers

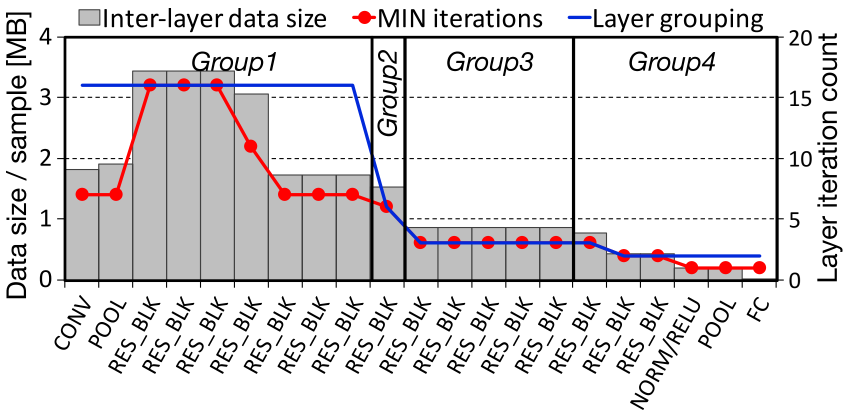


Figure 3.3: Per-block inter-layer data size, required layer iterations, and MBS layer grouping for ResNet50 with 32 samples.

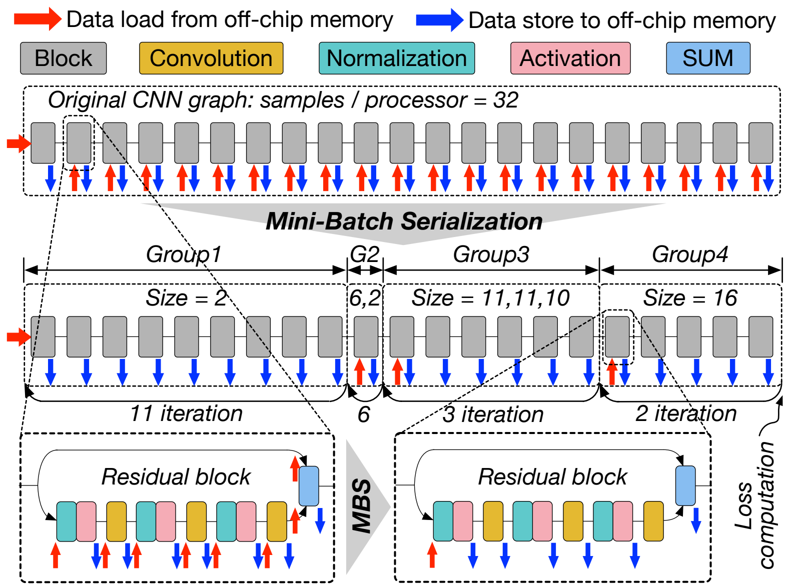


Figure 3.4: ResNet50 training flow by baseline and MBS.

that require the same number of sub-batch iterations. This is shown in Figure 3.3 where grey vertical bars represent the data volume required for the inter-layer data per layer (or one multi-branch module *block*) of ResNet50, and the red line represents the resulting minimal sub-batch iteration count for each layer. Then, layer groups are merged to improve overall locality: groups are merged by reducing the sub-batch size of one group to that of an adjacent group. The first group then requires more iterations (with more weight and gradient accesses), but inter-layer reuse increases across the two layers where the groups meet. The resulting grouping for this optimization for ResNet50 is shown with the blue line in Figure 3.3.[2] The mini-batch is then processed

[2]I also experimented with an optimal grouping of layers using exhaustive search, which improved traffic and performance by roughly 1% compared to my greedy optimization.

in several sub-batch iterations ($\lceil \frac{mini-batch\ size}{sub-batch\ size} \rceil$) within each group as shown in Figure 3.4, which emphasizes how locality is increased and memory traffic reduced across features and weights.

Back Propagation. In back propagation, MBS optimizes locality for both newly computed results and for data reloaded from the forward path. For example, as shown in Figure 3.1, MBS reuses the reloaded gradients more than once. Furthermore, both convolution and ReLU layers use activations from the forward path. However, only the gradient of ReLU is needed, which is always 0 (for negative activations) and 1 (for positive); thus, MBS uses a single bit per ReLU gradient instead of a 16b. I also allocate buffer space for normalization layers to reuse their inputs to compute their gradient and loss. As in the forward pass, reuse in back propagation is made possible by MBS processing one sub-batch at a time.

Data Synchronization. MBS maintains the original synchronization points across the entire mini-batch. Therefore, MBS accumulates the partial gradients of all learning parameters across all sub-batches. This requires storing partial results to memory, which is not needed in the conventional flow. However, this overhead is dwarfed by the improved reuse of layer outputs, especially considering that deeper layers with large weights are iterated over only a few times.

3.2.3 Data Reuse Within a Multi-Branch Module

Figure 3.4 also shows how MBS applies the same sub-batch approach to a multi-branch residual module of ResNet50. Such multi-branch modules are common in CNN architectures and offer additional reuse opportunities. Both the main path and shortcut branch share an input, and when they merge, their outputs are summed. Therefore, the module inputs should stay on chip until both paths have consumed them, and the output of the shortcut branch should stay on-chip while the main path output is computed. MBS does this by provisioning buffer space based on the needs of multi-branch *blocks*, where a block includes all the branches that share split and merge points—MBS essentially treats such a block as a layer for optimizing locality.

Maintaining locality for such shared nodes leads to additional storage requirements. The per-sample size is calculated by Equation 3.1 where: D_{in} and D_{out} indicate the sizes of the main-branch input and output; $D_{shortcut}$ is the size of the shortcut path output; L is the number of layers in the main branch; and b and l represent a specific branch and layer.

$$\begin{aligned} \frac{Space}{Sample} &= \max_{1\leq b\leq 2,\ 1\leq l\leq L} D_{in}(b,l) + D_{out}(b,l) + D_{cond}(b,l) \\ D_{cond}(b,l) &= (b{=}1 \ \&\ l{\neq}1) D_{block\_in} + (b{\neq}1) D_{block\_out} \end{aligned} \tag{3.1}$$

Similarly, for inception modules [96, 95], the block input is reused between branches, and the concatenated block output is eventually reused in the following layer. Therefore, MBS keeps both the block input and output on chip while executing the branches. The space required is shown in Equation 3.2, where B indicates the number of branches in a module and other notation is

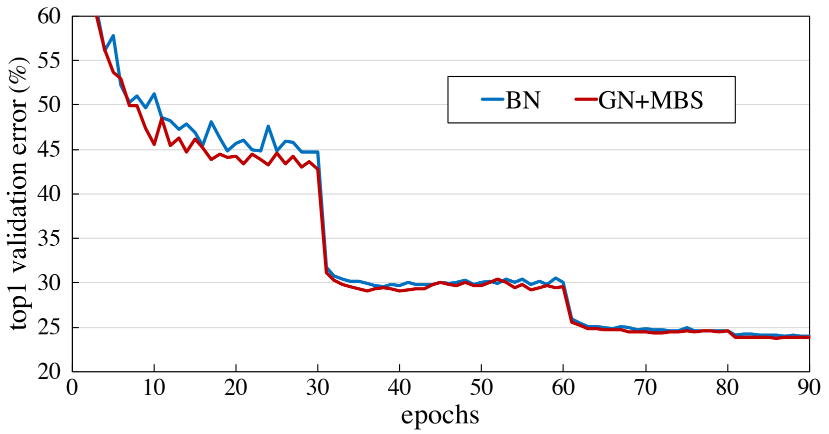


Figure 3.5: ResNet50 validation error trained with GN + MBS and BN.

as above.

$$\begin{aligned}\frac{Space}{Sample} &= \max_{1\le b\le B,\, 1\le l\le L} D_{in}(b,l) + D_{out}(b,l) + D_{cond}(l)\\ D_{cond}(l) &= (l \neq 1)D_{block\_in} + (l \neq L)D_{block\_out}\end{aligned} \tag{3.2}$$

3.2.4 Feature Normalization in MBS

While batch normalization (BN) is widely used in many modern CNNs, it is incompatible with MBS because BN requires many samples to work well and improve accuracy [46]—MBS cannot serialize computation if data across an entire mini-batch (per processor) is needed for normalization. Instead of using BN, I adapt group normalization (GN) [106] to MBS. GN normalizes across feature maps within a subset of channels in a single sample, as opposed to across an entire per-processor mini-batch. Thus, GN can be made compatible with MBS.

To use GN with MBS, the per-channel GN scale and shift parame-

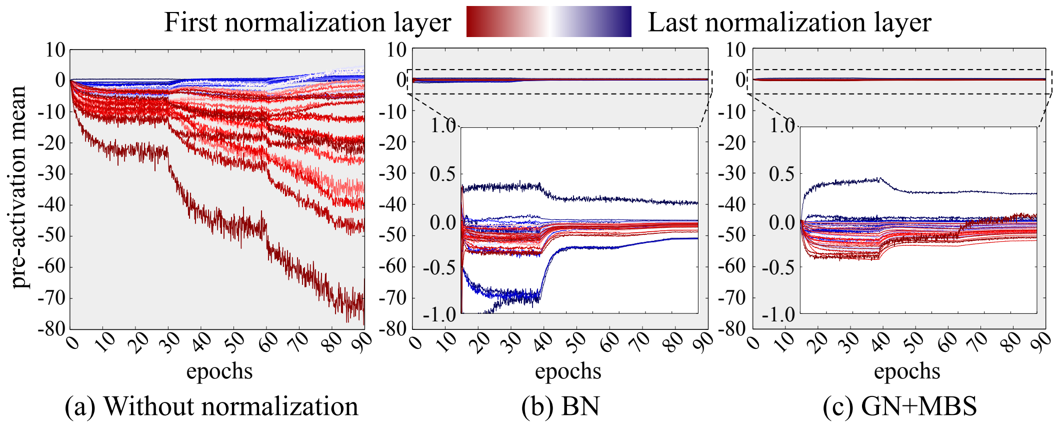


Figure 3.6: Pre-activation mean of each normalization layer. (*a*) **Without normalization,** (*b*) **BN ,(c) GN.** *b* **and** *c* **are zoomed in.**

ters must be re-fetched at every sub-batch iteration within a layer group. Additionally in backpropagation, the gradients of these parameters must be accumulated across all sub-batches just like the weights of convolution layers. However, since the size of these parameters is only two times the number of channels per layer, they can easily be stored in the on-chip buffer and incur no overhead.

I confirm previous results and demonstrate that both GN and BN provide comparable training effectiveness. Figure 3.5 compares the validation error curves with BN and MBS-GN when training ResNet50 on ImageNet [22]. Figure 3.6 shows that both MBS-GN and BN provide similar normalization, in that both have similar pre-activation (output of normalization) distributions across layers (unlike training without normalization). In this experiment, minibatch samples of 128 are distributed across 4 GPUs and an initial learning rate 0.05 used considering the linear learning rate scale rule [10, 81].

3.3 Efficient MBS Training on the WaveCore Accelerator

In this section, I introduce WaveCore, a CNN training accelerator for efficient execution of partially serialized training workloads. The design of WaveCore is based on prior training accelerators with large systolic arrays [51, 21]. To help understand its operation, I first describe the overall operation mechanism of a systolic array then introduce the detailed improvements I develop for WaveCore.

3.3.1 MBS Training on a Systolic Array

Like prior work, each PE in my proposed systolic array has mixed precision units: 16b inputs are multiplied with accumulation performed in 32 bits to reduce both computation and data traffic overheads [71]. Also like prior work [21], I use a 128×128 systolic array for high performance and to circumvent power delivery challenges.

A systolic computation is often divided into multiple *waves*, where each wave proceeds with inputs flowing toward outputs without any stalls or changes to the computational pattern. Between waves, it is sometimes necessary to let the pipeline through the array drain and then refill. This introduces idle time which reduces utilization and hence hurts performance and efficiency. Convolution and matrix operations have efficient systolic implementations that have little idle time if an entire mini-batch is processed together. However, MBS processes an often small sub-batch, which significantly reduces

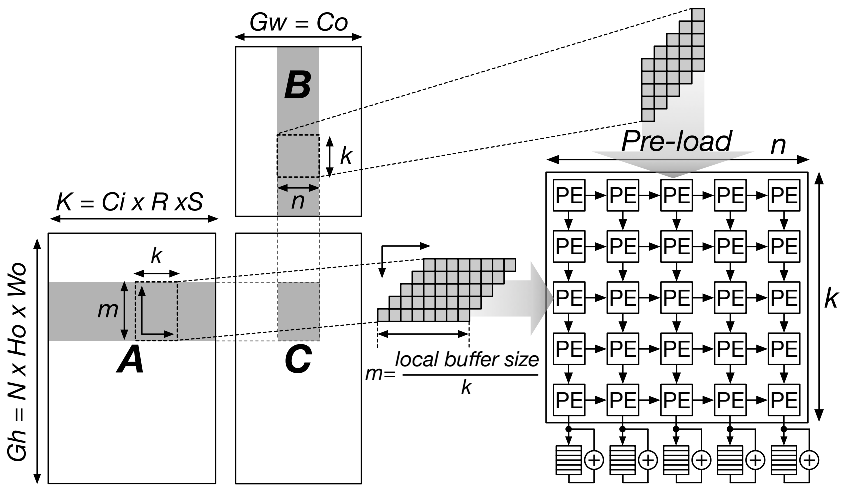
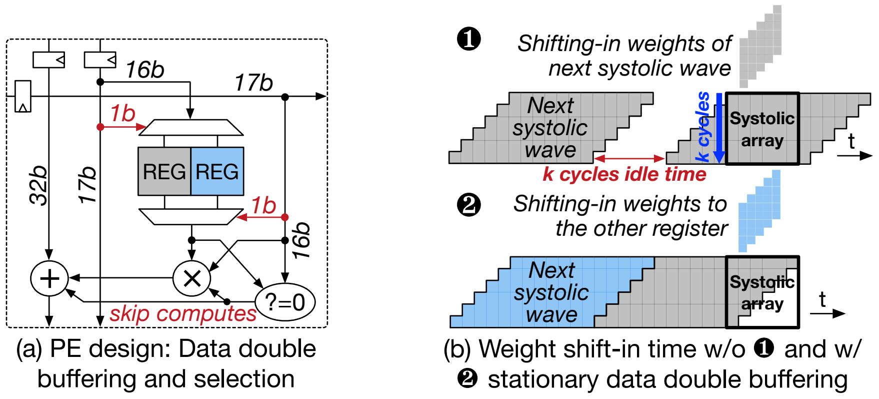


Figure 3.7: Tiled GEMM mapping to the systolic array of a convolution layer in forward propagation.



16b
17b
1b
REG
REG
1b
32b
17b
16b
?=0
skip computes
(a) PE design: Data double buffering and selection
❶ Shifting-in weights of next systolic wave
Next systolic wave
k cycles
Systolic array
t
k cycles idle time
❷ Shifting-in weights to the other register
Next systolic wave
Systolic array
t
(b) Weight shift-in time w/o ❶ and w/ ❷ stationary data double buffering

Figure 3.8: Removing inter-wave idle time by weight double buffering and control signal shift.

the utilization and performance of a conventional systolic array design. I address this challenge and maintain high systolic array utilization for MBS using a combination of two techniques.

Maintaining High Compute Unit Utilization with im2col. First, instead of directly mapping a convolution computation to a systolic array, I use the *im2col* algorithm described in Section **??**. I do this because efficient direct convolution on a systolic array requires tuning for every possible sub-batch size, which is difficult to do with the MBS approach that optimizes groupings to arbitrary size.

The major concern of processing GEMM on a large systolic array is that if the size of M, N, or K (the GEMM dimensions after im2col conversion) is smaller than the systolic array size, the compute units are significantly underutilized. However, based on my observation, the GEMM of each sub-batch constructed by im2col is large in all dimensions, and this maintains high systolic array utilization: early layers with small sub-batches have large feature maps and while the feature map sizes of later convolution layers are small, their large sub-batch size compensates.

Systolic Dataflow for Convolution GEMM. I block the convolution GEMM into multiple $m \times n$ tiles, which are processed in sequence through the array. Each tile corresponds to a portion of the output matrix (C). The width of each tile is equal to the width of the systolic array (n). The height (m) is chosen to maximize the size of a tile, thus minimizing the number of tiles per layer and improving utilization: $m = \frac{Local\ buffer\ size}{k = systolic\ array\ height}$. This is illustrated in Figure 3.7.

Each tile is processed using multiple waves through the systolic array, where each wave multiplies a block of input matrix A by a block of input matrix B. A block from B is first read one row at a time. Each row is shifted down until the array has one element of B per PE (this takes 5 cycles in the toy example of Figure 3.7). Then, a block of A is pipelined into the array with results for each element of C eventually accumulated at the bottom of the array as shown in the right side of Figure 3.7. Notice that in the figure, cycle 6 corresponds to the first row of the block of A having been multiplied and then accumulated by the first column of the block of B. In the following cycle, the second row of the A block completes its pipeline through the first column, while the first row of A now completes its dot product with the second column of the B block (and its output is at the bottom of the second column of the systolic array). Once a wave as described above completes, the next blocks of A and B are processed. As additional blocks are processed, their outputs are added to the current values of the C tile (a reduction across waves), eventually completing a tile of C in $\lceil K/k \rceil$ waves.

3.3.2 Gapless Waves with Per-Register Weight Double Buffering

The flow described above has one significant problem. Before every multiplication of blocks of A and B, the B block is read and distributed to the PEs, which requires k (PE array height) cycles (for reading and inter-PE shifting). No arithmetic occurs during these k cycles, which decreases performance (upper half of Figure 3.8b).

To remove this inter-wave idle time, I modify the basic PE design to double buffer weights (Figure 3.8a)—the next wave's weights are fetched and distributed into a second register within each PE while the current wave is still being processed. As the current wave starts draining from the PE array, the following wave starts immediately by feeding in a new block of A and multiplying by the second register that stores the next set of weights from B. Thus, there are no gaps between waves and an entire tile of C is computed without any idle time beyond the initial fill and final drain of the pipeline. In addition to the extra register in each PE, a minor further change is that a select signal for choosing which weight register to use is propagated along with the inputs of A and B. This optimization significantly boosts performance at very low cost: the simple 1b local signal between every two PEs and a 16b register and multiplexer between the two registers per PE. As in prior work, I also check for zero inputs and skip arithmetic in such cases to reduce energy consumption [77].

3.3.3 WaveCore Processor Architecture

In addition to the systolic cores, the WaveCore CNN training accelerator contains several more structures and units. Figure 3.9 illustrates the overall architecture of one core of the processor. There are two such cores in my proposed design that are connected by an on-chip network, similar to TPU v2 [21]. I describe these structures and estimate the area and power requirements of WaveCore below.

Local Buffers. Both A and B local input buffers are double-buffered. Double buffering enables the overlap of computation within the PEs with accesses to

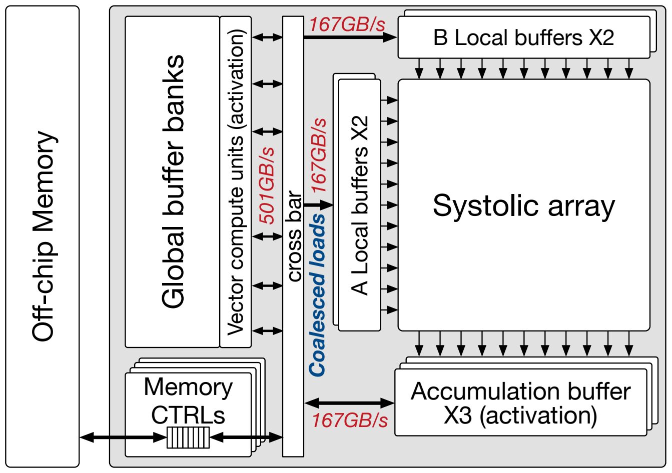


Figure 3.9: Per-core architecture of the WaveCore accelerator.

the global buffer and to memory and allows for very simple coarse-grain control of data transfers between buffers and memory. I choose the minimal size for each buffer, such that PEs never directly access the global buffer or memory, as this avoids access-related stalls. A half-buffer of B stores a 16b word for each PE and is thus 32KiB ($128 \times 128 \times 16b$). Each half-buffer for A is 64KiB because A blocks need to be twice as large as B blocks to avoid inter-wave idle time. The output accumulation buffer is triple-buffered because it holds the current output tile while the previous tile is being written to memory and the partial gradient sums for the next tile are read. Each part of this buffer holds an entire tile of C and is 128KiB. Note that while outputs are summed in 32b precision, the final write to the output buffer quantizes to 16b precision.

Global Buffers. The baseline global buffer is 10MiB and has 32 banks. This is sufficient for using MBS with modern CNNs and avoiding bank access conflicts. The global buffer is connected to all local buffers via a crossbar. To avoid duplicated data loads from the global buffer, I have memory load coalescing units that maintain high effective bus bandwidth utilization. My processor is designed to operate at a 0.7GHz clock frequency, and the data bandwidth of local and global buffers are set to fully support the systolic wave pipelining.

Main Memory. The off-chip memory is connected to memory controllers, which communicate with the on-chip buffers via the crossbar switches. The baseline WaveCore uses a single HBM2 stack with 4 dice [50], which provides 8GiB off-chip DRAM with 300GiB/s data bandwidth over 8 channels (4 channels per core). I choose HBM2 because it is used by other modern training accelerators [21, 76]. I later show that cheaper GDDR or even LPDDR memory can be sufficient for WaveCore.

Vector and Scalar Computing Units. The systolic array is used for convolutions and fully-connected matrix operations, but cannot be efficiently utilized by normalization, pooling, and activation layers, which require a relatively small number of arithmetic operations. Such layers are memory bandwidth bound, and I therefore process them using scalar and simple vector units that are placed close to the global buffer where their outputs are then stored.

3.3.4 Area and Peak Power Estimation

Area Estimation. I estimate the die area of WaveCore at 45nm technology and scale this estimate to 32nm to compare with other deep learning acceler-

Table 3.1: WaveCore accelerator specification and comparison to other training accelerators.

	V100	TPU v1	TPU v2	WaveCore
Technology (nm)	12 FFN	28	N/A	32
Die Area (mm^2)	812	≤ 331	N/A	534.0
Clock Freq (GHz)	1.53	0.7	0.7	0.7
TOPS / Die	125 (FP16)	92 (INT8)	45 (FP16)	45 (FP16)
Peak Power (W)	250	43	N/A	56
On-chip buffers (MiB)	33	24	N/A	20 (2×10)

ators Table 3.1. The estimated total area of the two-core WaveCore is 534.0 mm^2. I use a 24T flipflop design as reported in [54] and the floating point multiplier and adder designs reported in [40]. Each PE requires 12,173 um^2 and the multiplier and adder take more than 90% of the PE area. The estimated area of the 128×128 PE array is 199.45 mm^2, which accounts for 67% of WaveCore's area. The size of the global buffer and the vector compute units per core are estimated at 18.65 mm^2 and 4.33 mm^2, respectively. The crossbar has 24 256b-wide ports (32B memory access granularity). The area occupied by the network and the crossbar expands the chip width by 0.4mm, following the approach used to evaluate Dadiannao [17].

Power Modeling. I use a convolution layer that exhibits 100% systolic-array utilization to estimate the peak power consumption of WaveCore. WaveCore operates at 0.7GHz, which is the same as TPU v2 [21]. WaveCore consumes a maximum of 56W (Table 3.1). Here, I use one HBM2 chip as the off-chip memory and model its power using the Rambus power model [99] in 22nm technology. The SRAM buffer power is calculated with CACTI [12] configured for 32nm. The power consumed by multipliers and adders is taken from [32] and filpflops from [27]. The link and router power is calculated with Orion2.0 [53].

3.4 Evaluation

3.4.1 Evaluation Methodology

I evaluate the locality benefits of MBS and the performance and energy of WaveCore on three well-known modern *deep CNNs*: ResNet [37], Inception v3 [96], and Inception v4 [95]. I also evaluate a shallower CNN (AlexNet [57]) with few memory BW bound layers such as normalization and pooling. I use mini-batches of 32 samples per core (64 per chip) for the deep CNNs and 64 samples per core for AlexNet because of its smaller training context. I use 16b floating point for all CNNs with mixed-precision arithmetic (16b multiplication and 32b accumulation) [71].

For each network, I evaluate several execution configurations as summarized in Table 3.2: **Baseline** uses two-level GEMM input matrix blocking for effective data reuse within each convolution and FC layer [59]; **ArchOpt** adds weight double buffering for better PE utilization (all other configurations use ArchOpt), **Inter-Layer (IL)** reuses the shared data between layers but only when the per-layer memory footprint of the entire mini-batch fits within the on-chip buffer (i.e., not using the MBS approach), **MBS-FS** is naive MBS that fully serializes a mini-batch such that all layers in the CNN have the same sub-batch size, **MBS1** greedily forms layer groups to simultaneously op-

Table 3.2: Evaluation configuration description.

Configuration	Description
Baseline	2-level GEMM blocking
ArchOpt	Baseline + weight double buffering
IL	ArchOpt + inter-layer data reuse
MBS-FS	IL + serialize all layers using the same sub-batch size
MBS1	IL + greedy layer grouping
MBS2	MBS1 + inter-branch data reuse

timize both intra- and inter-layer data reuse, and **MBS2** additionally reuses the inter-branch data which requires different layer grouping than MBS1. I compare WaveCore with MBS to an NVIDIA TESLA V100 running Caffe [49] and report values averaged over 10 training iterations.

I develop a performance simulator for WaveCore and evaluate the performance, energy, and memory traffic of MBS. The WaveCore simulator accounts for all memory, buffers, and on-chip interconnect traffic as well as the arithmetic operations. The default WaveCore uses a single HBM2 chip with 4Hi stacks. I also scale memory bandwidth using two HBM2 chips to launch a larger mini-batch per accelerator (and to more closely match commercial accelerators). Because MBS significantly reduces memory traffic, I also evaluate lower-bandwidth main memory options that are cheaper and offer higher capacity (GDDR5 and LPDDR4). The off-chip memory configurations of WaveCore are listed in Table 3.3.

Table 3.3: Off-chip memory configuration.

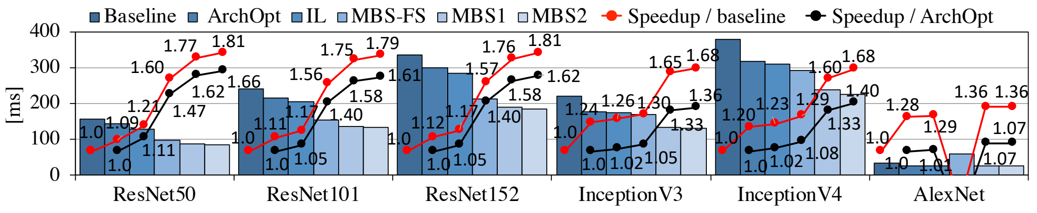
Memory type	Per-chip configuration	Chip #	Total BW
HBM2	300 GiB/s, 8 GiB, 8 channels	x1	300 GiB/s
HBM2$\times$2		x2	600 GiB/s
GDDR5	32 GiB/s, 1GiB, 1 channel	x12	384 GiB/s
LPDDR4	29.9 GiB/s, 2GiB, 1 channel	x8	239.2 GiB/s

3.4.2 Evaluation Results

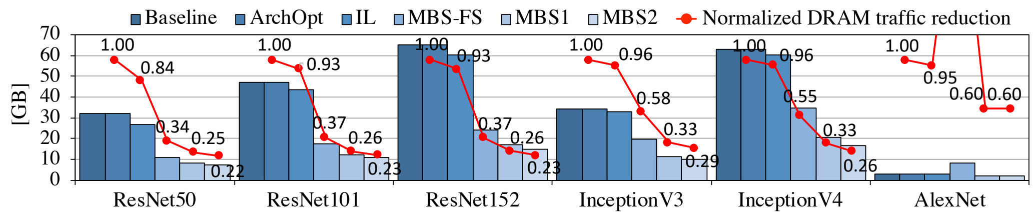
Figure 3.10 compares the per-training-step execution time, energy consumption, and DRAM traffic of my proposed technique. In each of the subfigures, bars show absolute values and lines show relative ones. I normalize execution time separately to both Baseline and ArchOpt to isolate the impact of the architectural and algorithmic contributions of WaveCore and MBS.

Compared to Baseline, ArchOpt improves performance by 9–28% across CNNs by removing the idle time between systolic waves. The gain is particularly large for AlexNet because AlexNet has mostly convolution layers with few memory-BW bound layers. Similarly, while not shown in the figure, ArchOpt provides more benefit with MBS because the large reduction in memory traffic increases the relative impact of idle compute time. For example, MBS2 without ArchOpt is only 19% faster than Baseline on average, whereas the speedup is 67% for MBS2 with ArchOpt, a 48% improvement on average. ArchOpt has little energy benefit ($\sim$ 2%) because it conserves only static energy.

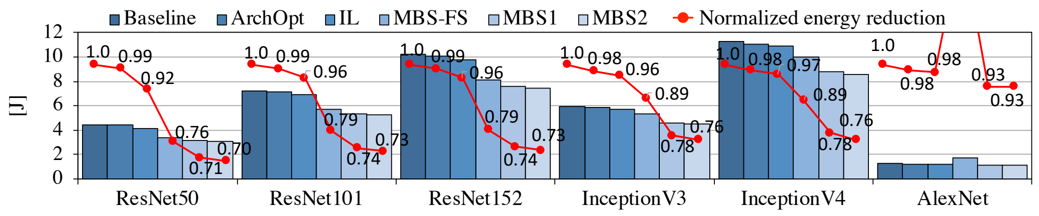
Inter-layer (IL), which is similar to prior locality approaches used for inference, has only a modest impact on performance and traffic because many layers have large footprints that exceed the buffer size.



(a) Execution time per training step. Speedups are normalized to Baseline and ArchOpt respectively



(b) Energy consumption per training step. The energy saving rate is normalized to Baseline



(c) DRAM traffic per training step. The traffic reduction rate is normalized to ArchOpt

Figure 3.10: Performance, DRAM traffic, and energy consumption sensitivity to the proposed network architecture reconfigurations and HW architecture optimization methods.

MBS-FS, which uses a single sub-batch size (and thus a single group) substantially reduces DRAM traffic (42–61%) for the deep CNNs because it utilizes inter-layer locality well. However, with a small sub-batch size, the time needed for the extra reads and writes of weight gradients used to accumulate them across sub-batches cannot be hidden, which reduces performance. This is evident in the performance trends of Inception v3 and v4, where MBS-FS is worse than IL. AlexNet exhibits a much larger performance loss with MBS-FS because it has three FC layers with large weights and the extra weight reads increase main memory traffic by 2.6×.

MBS1 balances inter- and intra-layer reuse and achieves large improvements in performance (33–54%) and DRAM traffic (67–71%) for the deep CNN compared to ArchOpt. AlexNet shows smaller gains as it lacks memory-BW bound layers. MBS1 also shows 22–26% energy saving for the deep CNNs compared to Baseline by reducing the DRAM energy portion from 21.6% to 8.7%. As WaveCore skips multiplication and addition when one of the inputs to a PE is zero, the contribution of DRAM traffic reduction to the overall energy saving is high. It is important to note that global buffer traffic is increased by a similar amount as DRAM traffic is decreased. However, there is still a large net energy saving because a global buffer access energy is 8× lower than that of DRAM.

MBS2 reduces DRAM traffic by an additional 7–10% and improves training performance by up to 5% compared to MBS1. MBS2 needs additional global buffer space to store the data at the shared multi-branch nodes, so the number of sub-batch iterations is larger than with MBS1. While more iterations imply a larger overhead for re-reading weights and gradients, the traffic saved by the reuse between branches is greater. The gain is slightly bigger for Inception v3 and v4 because the Inception modules have more branches and reuse opportunity scales linearly with the number of branches.

In summary, the highly-optimized MBS2 improves DRAM traffic by 71–77%, training performance by 36–60%, and energy consumption by 24–29% for the deep CNNs.

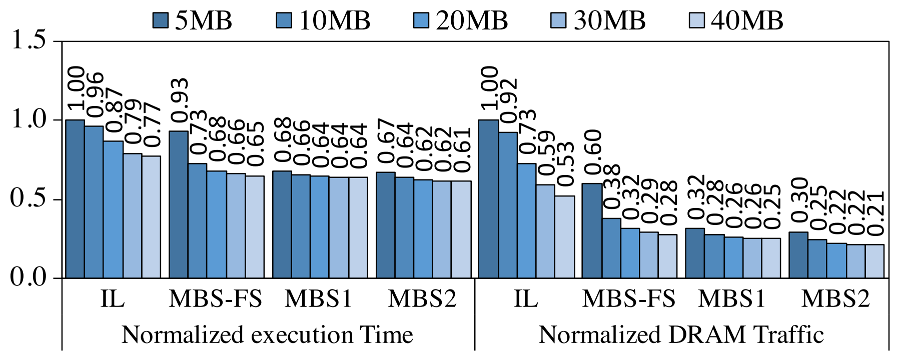


Figure 3.11: Memory traffic and performance sensitivity of ResNet50 to the global buffer size (Normalized to IL with 5MiB).

3.4.2.1 Sensitivity to Global Buffer Size

Another benefit of MBS is its low sensitivity to on-chip storage capacity. To showcase this, I compare the execution time and DRAM traffic per training step of ResNet50 for different configurations with different global buffer sizes (Figure 3.11). The per-core global buffer size is scaled from 5MiB to 40MiB and execution time and traffic are normalized to IL at 5MiB (ResNet's MBS scheduling requirement is smaller than 5MiB). Even with a 40MiB global buffer, only 47% of DRAM traffic is saved by IL; MBS2 saves 1.5X the traffic even with a 5MiB buffer. IL with 40MiB also provides less performance benefit than both MBS1 and MBS2 at just 5MiB. Both MBS1 and MBS2 show little performance and DRAM traffic variation for different buffer sizes because they simultaneously balance both intra- and inter-layer reuse. In contrast to the optimized MBS1 and MBS2, MBS-FS again suffers from the impact of reading and writing gradient partial sums.

3.4.2.2 Sensitivity to DRAM BW

Figure 3.12 highlights the ability of MBS to enable high performance even with lower-cost, lower-bandwidth memories. The figure compares the per-step training time of different configurations using various memory types (speedup is normalized to Baseline with 2×HBM2). The bandwidth of GDDR5 and LPDDR4 is 64% and 40% that of HBM2×2, respectively. While all implementations suffer from decreased bandwidth, the improved locality with MBS2 makes it far less sensitive with only a 4% performance drop when using off-package GDDR5 and a $< 15\%$ drop with low-cost LPDDR4. In this experiment, the off-chip memory space has been increased to 16GB to train 64 samples per core (128 per WaveCore) because off-package memories offer higher capacity.

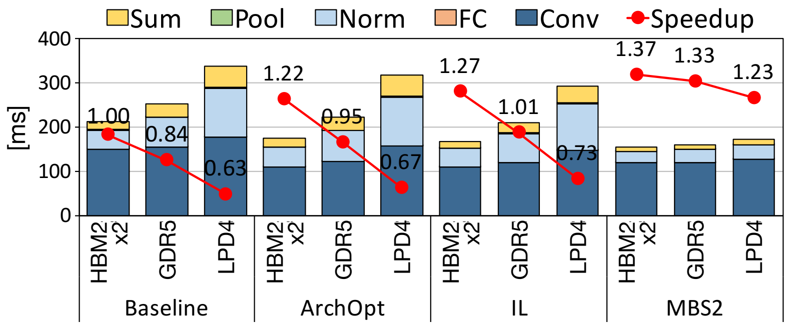


Figure 3.12: ResNet50 training performance sensitivity to the off-chip memory bandwidth and the execution time breakdown by layer.

3.4.2.3 Performance Comparison to GPU

Figure 3.13 compares the measured execution time per training step of an NVIDIA V100 GPU with my estimates for WaveCore with different DRAM configurations. Although a single WaveCore has 30% the peak compute and 27% the memory bandwidth (LPDDR4) of V100, it still exhibits better training performance. The performance gap widens as the network depth increases because many layers with low data parallelism cannot efficiently utilize the wide compute resources of the V100 GPU.

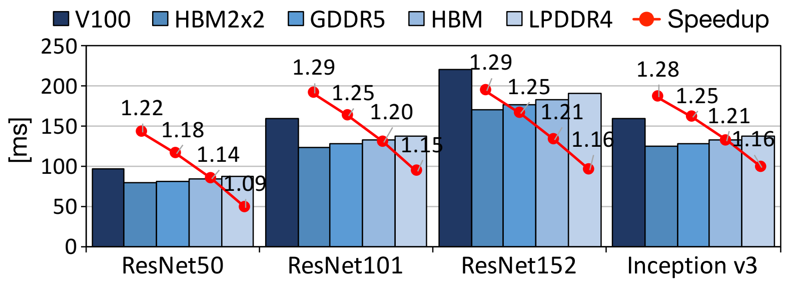


Figure 3.13: NVIDIA V100 GPU performance comparison to WaveCore + MBS2 with different memory types.

3.4.2.4 Systolic Array Utilization

As MBS propagates only a fraction of the mini-batch for each sub-batch iteration, it is important to observe its impact on the systolic core utilization. Figure 3.14 compares the utilization of convolution and FC layers for different CNNs. To isolate the impact of sub-batch size and the parallelism it makes available on utilization, this experiment uses unlimited DRAM bandwidth. Baseline suffers from low core utilization (average of 53.8%) due to the inter-wave idle time. Double buffering with ArchOpt increases the average utilization to 81.5%. MBS-FS exhibits lower utilization (66.7%) because the sub-batch size is determined solely by the large early layers. Optimizing reuse with different sub-batch sizes across layer groups with MBS1 and

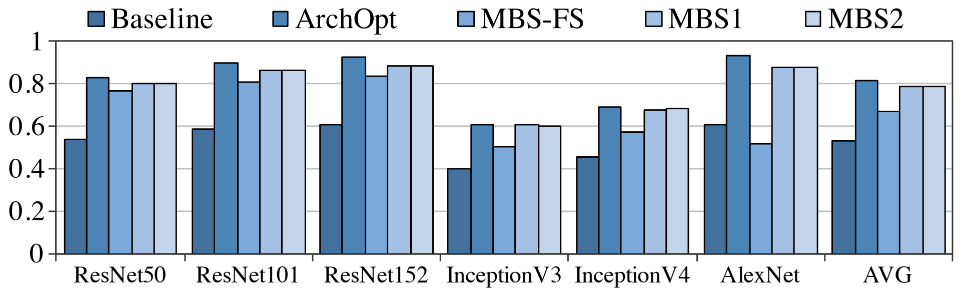


Figure 3.14: Systolic array utilization of different CNNs.

MBS2 regains the lost utilization and brings it up to 78.6%, within 3% of a full mini-batch. This small difference is largely a result of a few early layers with small channel counts, which result in particularly narrow tiles that do not fully utilize WaveCore's 128×128 systolic array. Later layers exhibit almost 100% utilization.

3.5 Related work

3.5.1 Inter-layer Data Reuse in Inference

No prior work has addressed locality-optimizations for CNN training. Instead, I discuss methods proposed for inference accelerators. Most inference accelerators optimize CNN scheduling to better utilize intra-layer locality [28, 15, 67, 14, 24, 51, 26]. They mainly focus on the data flow within a processing array, reducing data re-fetches by unrolling, or optimizing data access patterns within a convolution layer.

SCNN [77] is a scheduling method and architecture that reuses inter-layer data in CNN inference. SCNN uses the on-chip buffer to hold both the input and output features of each layer along with all weights. This is possible with a reasonable on-chip buffer size because SCNN relies on the fact that inference uses a single sample (mini-batch of 1), that features between layers are sparse because of ReLU, and that weights are even more sparse because they are pruned once training is complete. Together, an entire network can fit within an on-chip buffer.

However, the SCNN approach cannot be used for training because the same conditions do not hold true: mini-batches are large resulting in layer outputs that exceed buffer size, convolution layer outputs are not sparse, and weights are not sparse before pruning [33].

3.5.2 Layer Fusion

Fused-Layer CNN [5] is an inference flow that also utilizes inter-layer data. The approach is to divide the initial input to the CNN (the input feature map) into tiles and propagate one tile through multiple layers. Each convolution layer uses its input tile to produce a smaller output tile (because output cannot be produced for bands along the tile edges). The overlap between tiles is exploited via dedicated caches. While effective for the networks evaluated in [5], Fused-Layer CNN can not be applied to training modern deep CNNs, because: (1) convolution layers with small feature maps and large channel counts and weight data (deeper layers in modern CNNs) do not exhibit sufficient inter-tile locality; (2) normalization layers are incompatible with the tiling used for the depth-first propagation; (3) the inter-layer communication pattern in multi-branch modules, as well as in back propagation, is not only a direct communication between one layer to its following one; and (4) tiles shrink as they are propagated depth-first through the network, which limits available parallelism and likely hurts PE utilization.

3.5.3 Compiler Techniques for Graph Scheduling

To improve the intra- and inter-layer data reuse, prior work proposes network model scheduling techniques that capture data dependency and map to existing hardware to better utilize their resources avoiding redundant memory accesses [9, 87]. In particular, the technique proposed by Rotem et al. [87] analyzes the data dependency between layer operators in both high- and low-level representations to accommodate the target-device specific optimizations. However, unlike MBS, their techniques do not involve the interplay between scheduling optimization and hardware design. Also, MBS removes the scheduling constraints imposed by the algorithm choice (e.g. feature normalization) and enables multi-GEMM layer fusion, which is not feasible to such compiler-based techniques.

3.6 Discussion

3.6.1 Feature Normalization for CNNs with Few Channels

GN (group normalization) is a key algorithm that makes MBS training works as discussed in Section 3.2.4. However, GN requires many channels and channel groups for stable normalization [106]. For this reason, GN cannot be used to small models with a few channels such as CNN models for CIFAR datasets. To enable MBS training for small models, I propose **SBN (sub-batch normalization)**. Like BN, SBN normalizes features within batch and feature map dimensions but it uses the features of only a sub-batch thus avoids communication across different sub-batch propagations.

Figure 3.15 compares the top1 validation loss of ResNet32 with SBN trained on both CIFAR10 and CIFAR100 to the training results of the baseline with BN. I use two SBN configurations in this experiment: given that ResNet32 has three residual block stages with each sharing the same node, SBN16:32:64 uses sub-batch sizes of 16, 32, and 64 at each stage. This is 8, 4, and 2 times smaller batch sizes compared to the baseline mini-batch size of 128. The other SBN configuration uses SBN8:16:32 and this uses half the samples at each stage compared to SBN16:32:64. For these two configurations, SBN shows lower or similar validation loss compared to BN. This indicates that, with SBN, MBS can be effectively used for a small CNN model with just a few channels. SBN works only when sub-batch sizes of early layers are sufficiently large (e.g. 16 or 32). Thus, ResNet for ImageNet does not work with SBN and shows high accuracy drop.

3.6.2 Different Core Designs for Matrix Operations

I use a systolic array with input-stationary dataflow as the baseline core design. However, different core designs (e.g. different systolic dataflows or data distributions [16]) can be used as the baseline for MBS training. Although dif-

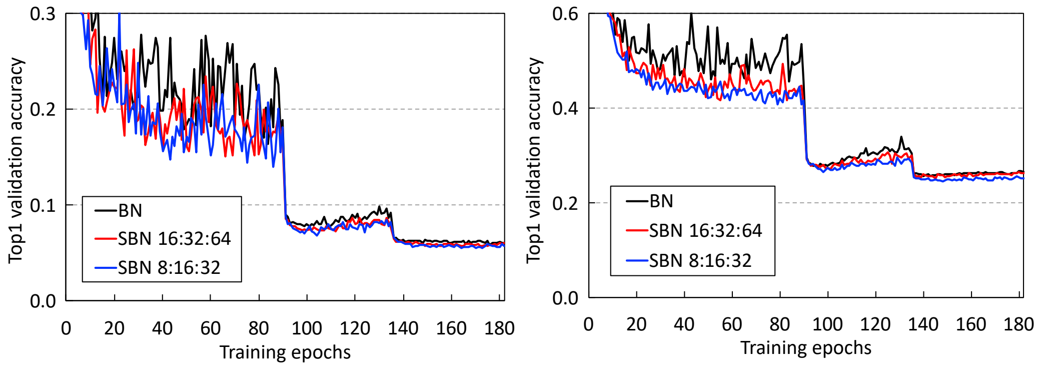


Figure 3.15: MBS training loss ResNet32 with SBN trained on (left) CIFAR10 and (right) CIFAR100.

ferent core designs need changes in resource allocation and component design, as MBS is a layer-wise data locality technique, its data reuse benefits remain the same.

I take a systolic array with output-stationary dataflow as an example to explain the changes needed to WaveCore for MBS training. Compared to the input-stationary dataflow Figure 3.7, where each PE shifts a partial sum to its adjacent PE at the bottom, output-stationary dataflow maintains the partial sum at each PE until the full accumulation is complete (Figure 3.16). After accumulation, the GEMM output is shifted out and this does not require the extra accumulator in the output buffer the like input-stationary dataflow. Since both features and weights are reused only while getting shifted between PEs in the array, the input reuse is lower compared to the input-stationary flow when the feature buffer size is large. Instead of input register double buffering at each PE, output register double buffering should be used at each PE to avoid inter-accumulation PE idling.

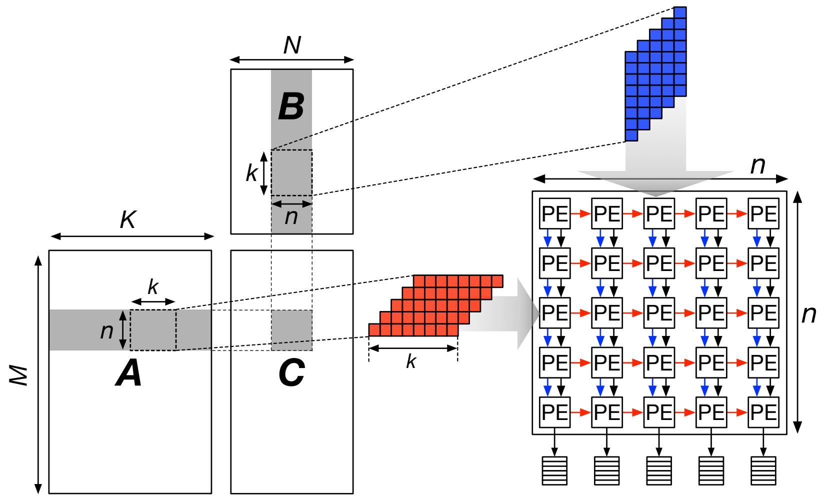


Figure 3.16: Input blocking and core mapping of output-stationary systolic array dataflow.

Other types of dot product engines with input broadcasting (e.g. SCNN [77] and Eyeriss [15]) can also be used as the baseline core design for MBS training. However, they are typically used in inference engines with relatively small number of PEs as input broadcasting is not scalable to the number of PEs compared to systolic input shifting.

3.6.3 Resource Pipelining using Fine-grain Layer Fusion

Instead of executing each layer function in bulk-synchronous fashion, it is also possible to pipeline the operations of vector layers (e.g. feature normalization and sum layers) with GEMM layers such that the vector operations are executed as soon as their inputs are ready. This enables better compute resource utilization. Once each tiled GEMM accumulation is finished, the output is first stored at the global buffer waiting for all the inputs belonging to the same group normalization granularity. Once all inputs are ready, the SIMD lanes perform the data reductions and affine transformation needed for feature normalization. Other vector operations that do not need reduction (e.g., element-wise sum and multiply) are executed when enough inputs to fill the SIMD lanes are ready.

This fine-grain multi-layer fusion can remove many redundant memory accesses between layers, and overlaps the computation time and memory accesses of vector operation layers. However, this fine-grain layer fusion presents a challenge to GEMM layer input blocking. This is because the reduction dimension of a vector layer followed by a GEMM layer is different from the optimal GEMM blocking dimensions. Vector layers (e.g., feature normalization) reduce inputs across the batch dimension, thus they need inputs from across the entire mini-batch. However, desired GEMM layer input reuse blocks GEMM tiles in the channel dimensions. Thus, processing the GEMM tiles in the batch dimension first reduces input reuse and causes unnecessary DRAM accesses.

On the other hand, MBS training is a good fit for fine-grain pipelining because it already fuses layers within a layer group in terms of memory accesses. When processing a convolution GEMM, one of its two input matrices is already buffered on chip, which enables flexible GEMM blocking with lower sensitivity to memory BW.

3.6.4 Balancing Matrix and Vector Computing Units

WaveCore contains a pipelined vector computation unit and its throughput is matched with the global buffer load bandwidth. This architecture is based on the layer-wise network execution flow so the vector compute throughput is set to consume the maximum data bandwidth and does not depend on the throughput of systolic arrays. However, if the layers with vector operations and GEMM operations are fused, the vector unit throughput should be set such that the computation time of vector operations is hidden by GEMM execution time to maximize GEMM compute unit utilization.

Based on my analysis, 98-99% of ResNet50 and Inception v4 are GEMM FLOPs from convolution layers and fully-connected layers. This explains the need for dedicated GEMM compute cores for CNN training accelerators. A 50:1 ratio between GEMM and vector compute units can roughly support high GEMM compute unit utilization. In detail, since pooling layers reduce the size of features of later layers (fewer vector FLOPs) but maintain the same total FLOPs, early layers need higher vector compute throughput than later layers. Therefore, using the average vector compute throughput (1/50 of GEMM compute throughput) can stall GEMM computation and underutilize expensive GEMM compute units. Two options can alleviate this GEMM core stalls: (1) the vector compute throughput can be set using the GEMM and vector compute FLOPs ratio of early layers, or (2) using a large on-chip buffer to hold more GEMM outputs to reduce the sensitivity to the throughput ratio between GEMM and vector computations.

3.6.5 Applying MBS to Other DNN Models

The memory traffic saving of MBS comes from reusing inter-layer data by partially serializing the training network propagation. CNN models are a good fit for MBS training because their model size is much smaller than the inter-layer data. Thus, serializing a CNN training iteration saves significant memory BW due to inter-layer data reuse. Also, the BW overhead from re-fetching weights between sub-batch iterations is negligible.

On the other hand, applying MBS training to other DNN models has little benefit because their per-layer model sizes tend to be larger than the size of inter-layer data. For example, NLP (neural language processing) [89, 82, 68] typically use RNN (recurrent neural network) [30, 6] models or self-attention based models [98, 23]. For both types of NLP models, the size of inter-layer data per sample is almost equal to, or smaller than its model size, thus serializing the training graph gives no benefit. In addition, the layers of such models maintain similar activation sizes across layers, unlike CNN models that gradually decreases the feature dimension using pooling. Thus NLP models do not need the inter- and intra-layer reuse balancing of MBS.

3.7 Summary

In this chapter, I introduce MBS, an approach to reuse the inter-layer data in CNN training and balance its locality with that of intra-layer data. MBS reconfigures the CNN computation graph by partitioning a mini-batch of samples into multiple sub-batches whose memory footprint fits within on-chip storage. I show that MBS reduces the volume of DRAM accesses by up to 74% while providing a high processing-element utilization of 79%. Additionally, I introduce a technique to exploit data reuse opportunities between branches in CNN multi-branch Residual and Inception modules.

To efficiently use MBS CNN training, I introduce WaveCore, a systolic-array based CNN training accelerator. I design WaveCore in a way to double-buffer data within its processing elements to remove idle time between the systolic waves used to compute the convolution and fully-connected layer outputs. WaveCore also triple-buffers to support concurrent execution of the original tiled GEMM and the partial SUM reduction needed by MBS training. My evaluation demonstrates that I expect single WaveCore with MBS to achieve higher performance than one V100 GPU despite having the GPU having 3× higher peak performance and memory bandwidth.

Furthermore, the high locality MBS achieved by balancing intra- and inter-layer reuse makes WaveCore very robust to memory design decisions. I demonstrate that both on-chip buffer capacity and available off-chip bandwidth have far smaller impact than using a conventional training approach. In particular, unlike to the conventional mini-batch training and naive serialization that fail to capture inter-layer locality with small on-chip buffers, WaveCore with MBS training highly reuses inter-layer data using only 5MB on-chip buffers. Also, even with a low-cost LPDDR4 DRAM system (the same DRAM used for mobile phones), WaveCore can outperform a high-end V100 GPU.

Chapter 4

Fast Model Training with Dynamic Sparse Model Reconfiguration

This chapter discusses fast CNN model training that structurally continuously prunes learning parameters from the beginning of training. I first explain the proposed training algorithm and show the observation that supports my motivation for pruning while training. Then, I introduce three key algorithmic techniques for the practical implementation of the proposed training mechanism. I implement my ideas using Pytorch and evaluate different CNN models for CIFAR and ImageNet datasets with multiple NVIDIA GPU types.

[1]

4.1 Motivation for Continuous Model Pruning During Training

Training a modern CNN requires millions of computation and memory bandwidth-intensive iterations. To reduce this high complexity of training and

[1]This chapter was published in International Conference for High Performance Computing, Networking, Storage and Analysis (SC) in November 2019. Sangkug Lym contributed to this work as the main author.

eventually the training time, I propose to use model pruning during training. Model pruning is a commonly used technique to make network inference fast and energy-efficient. Although most model pruning mechanisms are focused on inference tasks, I find that pruning can also substantially accelerate training.

To realize training speedup using model pruning, first, unimportant parameters should be identified and pruned during training. However, most prior pruning methods start pruning from a pre-trained model and repeat the process of pruning and retraining to remove unimportant parameters [39, 72, 112, 38]. Since such trial-and-error model pruning has to first train the baseline model, it eventually increases overall training time. Second, after each pruning, the pruned model should have a dense form to avoid complex data indexing for efficient acceleration on data-parallel accelerators. However, most prior work prunes individual parameters to match the goal of maximally reducing model size [3, 18]. The models pruned with such an approach perform poorly on common training accelerators with high compute throughput because they require complex indexing [103].

In contrast, structured pruning can maintain accelerator-friendly pruned models. However, prior work that structurally prunes parameters during training is scarce. In particular, Wen et al. [103] propose *SSL*, a pruning approach that sparsifies weights while training a CNN, but does not remove them from the model until training is complete. They use parameter regularization and prune parameters at channel granularity. The pruned models then exhibit high performance with training accelerators. They maintain the original dense network architecture until the end of training because sparsified weights may *revive* later in training thus not improving performance. Furthermore, in their evaluation, they start pruning from a pre-trained model thus SSL actually requires more time to train than baseline, because it first trains the dense baseline and then prunes it.

The pruning approach proposed by Zhou et al. [113] prunes the zeroed parameters during training but does not reconfigure the network architecture. Instead, gradient updates in back-propagation are skipped by setting weight momentum to zero. Since this mechanism still performs all training computation, no training performance improvement is achieved. Alvarez and Salzmann [4] propose to reconfigure the sparse network architecture and reload the model to accelerate training. However, their training method prunes a CNN model only once during training, which limits its training performance gains.

Opportunities in Pruning During Training. In this dissertation, I present **PruneTrain**, a structured model pruning mechanism for effective training speedup. PruneTrain continuously prunes channels of a CNN model and reconfigures the network architecture into a new and smaller dense form during training. This continuous pruning and reconfiguring can significantly speed up training for two reasons. First, of all the convolutional channels that regularization sparsifies, most are sparsified very early in the training process, so pruning these channels has a significant positive impact on the overall training time. Second, regularization sparsifies the channels gradually over time, so it is more beneficial to prune the sparsified channels frequently, as opposed to

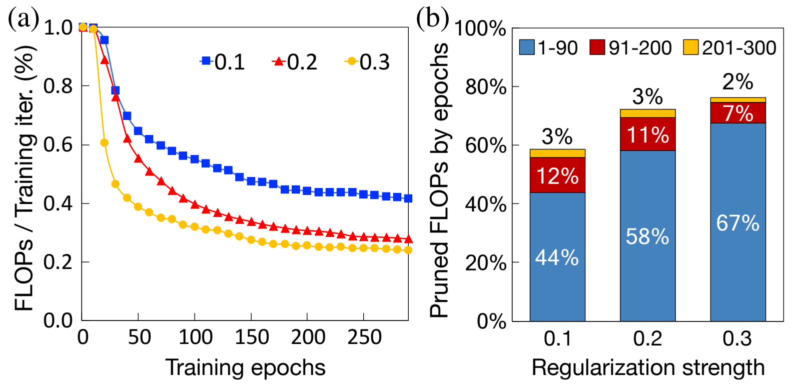


Figure 4.1: (a) FLOPs per training iteration normalized to the dense baseline (ResNet50 on CIFAR10). (b) Breakdown of prunable training FLOPs over epochs.

pruning them only once.

To show this, I train ResNet50, one of the most commonly used image classifiers for various vision applications [66, 35, 65], on the CIFAR10 dataset with regularization. Every epoch, I measure the FLOPs (floating-point operations) per training iteration, assuming I can prune the unnecessary channels every 10 epochs. Figure 4.1a shows the FLOPS per iteration normalized to the dense baseline. Each line in the figure shows the FLOPs using a different regularization strength. I will describe the definition of regularization strength in Section 4.2. Regardless of the strength, the majority of FLOPs are pruned in the early epochs, with the rate of pruning gradually saturating. This is further shown by the breakdown of aggregated pruned FLOPs (Figure 4.1b) over three training phases, where most FLOPs are pruned within the first 90 training epochs. This means continuous pruning and reconfiguration can save a large fraction of computation for the rest of the training.

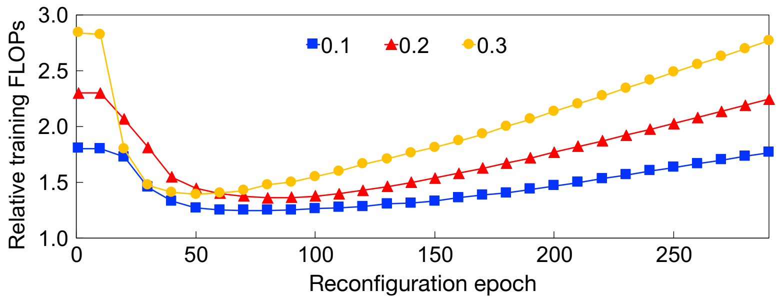


Figure 4.2: Training computation overhead of one-time network reconfiguration at different training epoch compared to PruneTrain; each line in (a) **and** (c) **is the result of different sparsification strengths.**

Also, given that weights are gradually sparsified during training, it is apparent that continuous and timely model pruning and reconfiguration can reduce training computations much more effectively than the one-time reconfiguration. Figure 4.2 compares the training FLOPs of one-time pruning and reconfiguration used in pruningior work [4] to PruneTrain. Regardless of the strength of group lasso regularization, even with the optimistic assumption that the best reconfiguration point is known, prior works uses more than 25% additional training FLOPs compared to PruneTrain. In reality, it is impossible to know the best reconfiguration time a priori, and thus, PruneTrain prunes and reconfigures the models periodically.

4.2 Model Pruning Mechanism

Baseline Pruning Mechanism. Like prior work [103, 102, 4], I use group lasso regularization to sparsify weights so that they can be pruned. Group lasso regularization is a good match for PruneTrain because it is incorporated with the training optimization and imposes structure on the pruned weights, which I use to maintain an overall dense computation. Group lasso regularization modifies the optimization loss function to also include consideration for weight magnitude. This is shown in Equation 4.1, where the left term is the standard cross entropy classification loss and the right term is the general form of the group lasso regularizer. Here f is the network's prediction on the input x_i, $\boldsymbol{W}$ are the weights, l is the classification loss function between the prediction and its ground truth y_i, N is the mini-batch size, G is the number of groups chosen for the regularizer, and λ_i are tunable coefficient that set the strength of sparsification.

$$\min_{\boldsymbol{W}} \left(\frac{1}{N} \sum_{i=1}^{N} l(y_i, f(x_i, \boldsymbol{W})) + \sum_{g=1}^{G} \lambda_g \cdot ||\boldsymbol{W}_g||_2 \right) \tag{4.1}$$

This lasso regularization sparsifies groups of weights by forcing the weights in each group to very small values, when possible without incurring high error. After sparsification, I use a small threshold of 10^{-4} to zero out these weights.

Proposed Group Lasso Design. I design a specific group lasso regularizer that groups the weights of each channel (input or output) of each layer. I also choose a single global regularization strength parameter λ rather than adjust the penalty per group. The resulting regularizer term is shown in Equation 4.2, where L is the number of layers in the CNN and C_l and K_l are the number of input and output channels in a layer, respectively.

$$\lambda \cdot \sum_{l=1}^{L} \left(\sum_{c_l=1}^{C_l} ||\boldsymbol{W}_{c_l,:,:,:}||_2 + \sum_{k_l=1}^{K_l} ||\boldsymbol{W}_{:,k_l,:,:}||_2 \right) \tag{4.2}$$

Prior work proposes to penalize each channel proportionally to its number of weights in order to maintain similar regularization strength across all channels [91, 3]. Instead, I choose to use a single global regularization penalty coefficient because this emphasizes reducing computation over reducing model size. All convolution layers of a CNN have similar computation cost. Because early layers have fewer channels and each channel has larger features, each channel of their layers involves more computation. Therefore, applying a single global penalty coefficient effectively prioritizes sparsifying large features, which leads to greater computation cost reduction. I do not apply group lasso to the input channels of the first convolution layer and the output neurons of the last fully-connected layer, because the input data and output predictions of a CNN have logical significance and should always be dense.

4.2.1 Regularization Penalty Coefficient Setup

To use lasso regularization from the beginning of training, the penalty coefficient λ should be carefully set to both maintain high prediction accuracy and to achieve a high pruning rate. I develop a new technique to set this strength coefficient without requiring resource-intensive hyper-parameter tuning. To do so, I choose λ using the ratio of group lasso regularization loss out of the total loss (the sum of the group lasso regularization loss and the classification loss). This *group lasso penalty ratio* is shown in Equation 4.3. Based on my observations of several CNN models (ResNet32/50 and VGG11/13) and training data (CIFAR10, CIFAR100, and ImageNet), I find that using a group lasso penalty ratio of 20-25% robustly achieves high structural model pruning ($> 50\%$) with small accuracy impact ($< 2\%$).

$$Lasso\, penalty\, ratio = \frac{\lambda \sum_g^G ||\boldsymbol{W}_{g,:}||}{l(y_i, f(x_i, \boldsymbol{W})) + \lambda \sum_g^G ||\boldsymbol{W}_{g,:}||} \tag{4.3}$$

I compute this using the random values to which weights are initialized at the beginning of training and the cross-entropy loss calculated after the very first network forward propagation. This penalty coefficient is set once at the first training iteration and maintained through training. Without my approach, prior work searches for the desired lasso regularization penalty coefficient, e.g., by trying random coefficient values until one that has a small impact on accuracy is found [4, 103]. This can potentially require many training runs for each CNN being trained and increase total training time.

4.2.2 Layer Removal by Overlapping Regularization Groups

Wen et al. [103] propose to use layer-wise lasso groups for regularization in order to remove layers of a CNN with short-cut connections. However, I do not include such grouping in my regularizer. I find that because there is an overlap in the weights between input and output channel lasso groups (Figure 4.3a), unimportant layers are eventually removed even without additional layer-wise weight regularization. As an example, when an input channel becomes sparse (Figure 4.3b) by lasso regularization, it gradually sparsifies all the intersecting output channels (c), eventually leading to the entire layer to become zero.

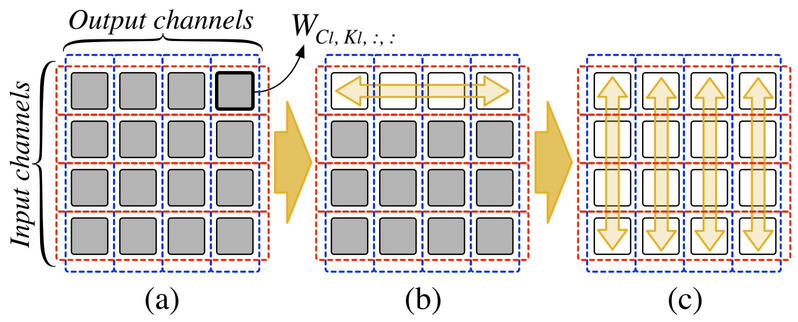


Figure 4.3: Group lasso regularization structure of a convolution layer: Weights of a filter (each square box) affect the sparsification of weights in both input and output channels (red and blue dotted boxes). The white filters are zeroed-out after sparsification.

4.3 Dynamic Network Reconfiguration

The main goal of PruneTrain is reducing the training cost and time by continuously pruning the spasified channels or layers and reconfiguring the network architecture into a more cost-efficient form during training. There are two main concerns with doing so. The first is that pruning while training might prematurely remove weights that are unimportant early in training but become important as training proceeds. The second, is that the overhead of processing a pruned network exceeds any benefits realized by training a smaller model.

4.3.1 Early Weight Pruning

A prior pruning mechanism for CNNs that uses group lasso regularization, *SSL* [103], maintains the sparsified channels until the end of training instead of removing them from the model. This is because pruning while training prohibits weights from "reviving" and becoming non-zero as training proceeds. This can happen as gradients flow back from the last FC layer and potentially increase the value of previously-zeroed weights. However, I observe that already-zeroed input and output channels of convolution layers are likely to suppress such revived weights from ever becoming large. This can be inferred from the equation of the local weight gradients for a layer l:

$$\frac{\partial L}{\partial \boldsymbol{W}_l} = \boldsymbol{z}_{l-1} \circledast \frac{\partial L}{\partial \boldsymbol{x}_l}^T \tag{4.4}$$

Here, $\circledast$ is convolution operator, and z_{l-1} and $\frac{\partial L}{\partial \boldsymbol{x}_l}$ are the input activations (or input features) and the upstream gradients from the subsequent normalization layer. If a channel is sparsified and zeroed-out, its convolution outputs x_{l-1} are zeroed and they remain zero after normalization and activation layers, meaning that z_{l-1} is zero. Also, if an input channel of the subsequent convolution layer (l) is zeroed, the upstream gradients of this input channel are forced to be small. Thus, the gradients after passing the normalization layer $\frac{\partial L}{\partial x_l}$ are also kept small by the gradient equation from [46]. Therefore, using Equation 4.4, the gradients of zeroed weights are forced to remain very small and often zero, effectively restricting the previously zeroed weights from reviving.

This behavior is apparent in Figure 4.4 that shows the output channel sparsity of three layers of ResNet50 [36] trained on CIFAR10 dataset across training epochs. Each point in the graph is the absolute maximum value among the parameters of each output channel. If the absolute maximum value of a channel becomes smaller than the threshold (10^{-4}), the parameters of the channel are zeroed out (white). Convolution layers 5 and 6 are typical and none of the weights from the zeroed output channels revive. Although some

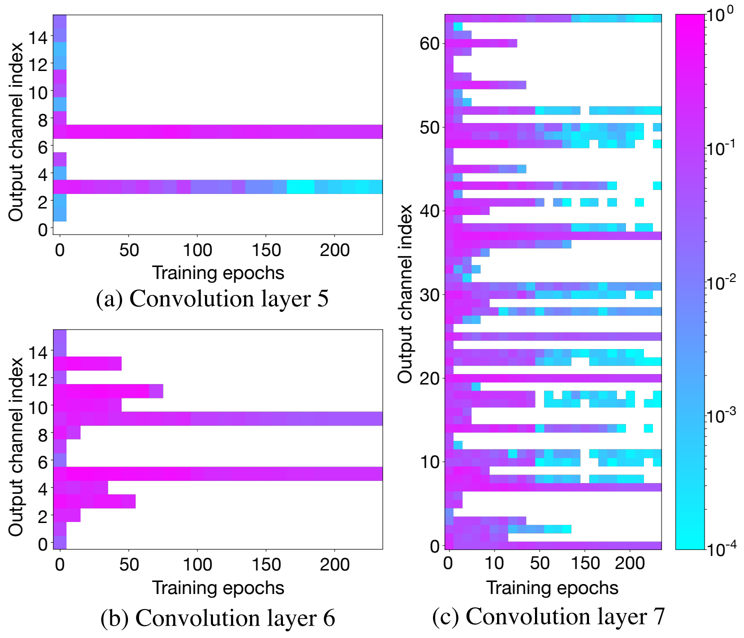


Figure 4.4: The maximum absolute weight value of each output channel over training epochs. Three convolution layers belong to one residual path of ResNet50 trained on CIFAR10.

parameters in output channels of convolution layer 7 revive, their weight values are still very small and near the threshold, indicating a very small contribution to the prediction accuracy of the final learned model. Similar patterns are observed in all convolution layers of different ResNet and VGG models on CIFAR10/100, with all layers exhibiting no significant revived parameters.

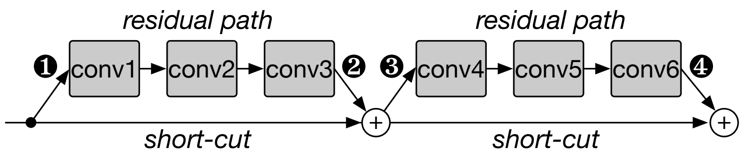
4.3.2 Robustness to Reconfiguration Interval

I now discuss the practical mechanisms for performing dynamic reconfiguration. I define a reconfiguration interval, such that after every such interval the zeroed input and output channels are pruned out. Note that if all the sparsified input and output channels are pruned, there is a possibility of a mismatch between the dimensions of the output channels of one layer and to the input channels of the next. To maintain dimension consistency, I only prune the intersection of the sparsified channels of any two adjacent layers. At any reconfiguration, all training variables of the remaining channels (e.g., parameter momentums) are kept as is.

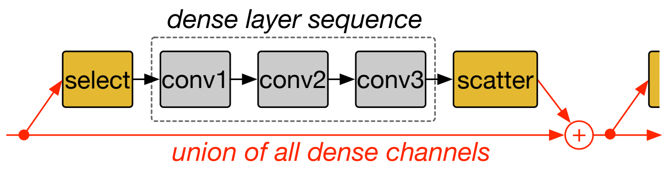
The reconfiguration interval is the only additional hyperparameter added by PruneTrain. Intuitively, a very short reconfiguration interval may degrade learning quality while a long interval offers less speedup opportunity. I extensively evaluate the impact of the reconfiguration interval in Section 4.4.4 and show that training is robust within a wide range of reconfiguration intervals.

4.3.3 Channel Union for CNNs with Short-cut Connections

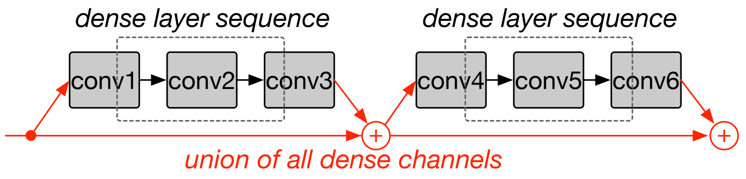
Short-cut connections are widely adopted in modern CNNs, including ResNet and its many variations [37, 107, 44, 45, 115, 83]. They enable deep networks by mitigating the vanishing-gradients problem and achieve high accuracy [36]. For such CNNs, the channels of the convolution layers at a merge-point should match in dimensionality after each reconfiguration for proper feature propagation (Figure 6.9a). I propose two mechanisms to ensure this occurs. The first is **channel gating** layers that add gating to each residual branch to match dimensions, as shown in Figure 6.9b. This ensures that all convolution layers in a residual block operate only on dense channels by



(a) Residual modules: the channel dimensionality of the convolution layers sharing the same node (❶, ❷, ❸, and ❹) should match.



(b) Channel gating: *channel select* and and *channel scatter* layers match the channels indexes.

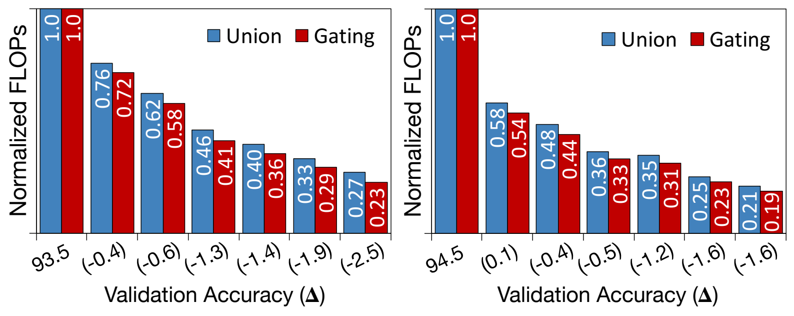


(c) Channel union: the first and the last convolution layers of each residual path contain sparse channels.

Figure 4.5: Channel indexing for CNNs with short-cut paths.

gathering and scattering the dense channel indices. This improves on the channel sub-sampling approach proposed by [39], with channel sub-sampling only avoiding redundant computation of the very first convolution layer of each residual block.

I evaluate channel gating on an NVIDIA V100 GPU and find that channel gating involves significant memory accesses for tensor reshaping needed for channel indexing that often slows down training. Therefore, as an alternative, I propose **channel union** that does not need any tensor reshaping and data indexing. Channel union prunes only the intersection of sparsified channels



(a) Inference FLOPs (ResNet32) (b) Inference FLOPs (ResNet50)

Figure 4.6: Normalized training and inference FLOPs of ResNet32 and ResNet50 on CIFAR10 with different pruning intensity.

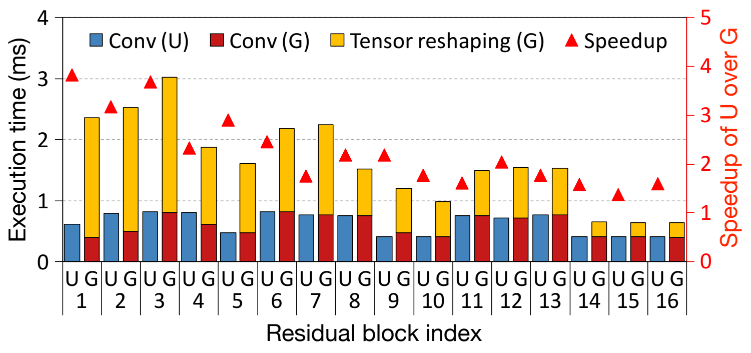


Figure 4.7: Per-layer execution time of channel gating and channel union for ResNet50 for ImageNet. G and U indicate channel gating and channel union respectively.

of all neighboring convolution layers within a residual stage (residual blocks sharing the same node). For instance, in Figure 6.9c, the union of the dense input channels of convolution layer 1 and 4 and the dense output channels of convolution layer 3 and 6 are maintained. As each following residual path adds new information to the shared node, the early convolution layers in the stage (convolution layer 1) have to process operations from the sparse channels, thereby performing redundant operations.

However, my experiments show that the additional FLOPs from channel union, as compared to channel gating are very small. Figure 4.6 compares the normalized inference FLOPs of channel gating and channel union for ResNet32 and ResNet50 pruned with different intensities. Across different pruning rates, the FLOPs difference is only 1-6%, but the overhead saved from indexing is substantial. Additionally, this FLOPs difference does not grow with increasing layer depth as shown in Figure 4.6 comparing ResNet32 and ResNet50. Figure 4.7 shows the measured per-layer (the last layer of each residual block) execution time of ResNet50 for ImageNet. For all residual blocks, channel union shows far less execution time compared to channel gating. Especially, the tensor reshaping time of early layers has bigger overhead as their activation size is eight times bigger than the layers in the last residual block.

4.3.4 Dynamic Mini-batch Adjustment

As discussed in Section 2.1, training with a large mini-batch reduces the frequency of costly inter-GPU communication and off-chip memory accesses for model updates. In addition, a larger mini-batch increases the data parallelism available at each network layer improving HW utilization. I find that the gradual channel and layer pruning simultaneously reduces available data parallelism and decreases the memory capacity requirement for training, thus allowing the use of larger mini-batches. The latter allows us to compensate for the former as follows.

I propose *dynamic mini-batch adjustment* to increase the size of the mini-batch by monitoring the memory context volume of a training iteration, which is gradually decreased by PruneTrain. When channels are pruned by PruneTrain, the output features corresponding to these channels are also not generated, which reduces the off-chip memory space required for back-propagation. In particular, early layers of a CNN have larger features and removing the channels of these layers effectively reduces the training memory requirement. Using a global regularization penalty, PruneTrain prunes the channels of early layers by a larger ratio than prior work and enables using a larger mini-batch over training epochs. At every network architecture reconfiguration, PruneTrain monitors the off-chip memory capacity required for a training iteration and increases the mini-batch size when possible.

However, dynamically increasing the size of the mini-batch alone does not guarantee high prediction accuracy as it is the hyperparameter closely coupled with the learning rate. To maintain the algorithmic functionality, I increase the learning rate by the same ratio of a mini-batch size increase to maintain the same learning quality. This mechanism is similar to adjusting the mini-batch size instead of decaying the learning rate, as proposed by Smith et al. [93] and Jastrzkebski et al. [47]. However, my proposed mechanism is different in that I change the mini-batch size and learning rate dynamically at any point during training, unlike this prior work that changes them at the original learning rate decay points. Note that dynamic mini-batch size adjustment relies on the linear relation between mini-batch size and learning rate. For other deep learning applications that have a different relation, an appropriate learning rate adjustment rule can be adopted instead (e.g., the square root scaling rule for language models [81]). I evaluate dynamic mini-batch adjustment by training ResNet50 on both CIFAR and ImageNet datasets and confirm that it maintains equally high accuracy compared to the baseline PruneTrain.

4.4 Evaluation

4.4.1 Evaluation Methodology

I evaluate PruneTrain on both small (CIFAR10 and CIFAR100 [56]) and large datasets (ImageNet [22]). I train four CNNs (ResNet32, ResNet50, VGG11, and VGG13) on CIFAR and ResNet50 on ImageNet, which is the most commonly used modern CNN for modern vision applications [66, 35, 65]. I use a mini-batch size of 128 and 256 (64 per GPU) for CIFAR and ImageNet training runs and a learning rate of 0.1 for both as the baseline hyperparameters [36]. I use four NVIDIA 1080 Ti and V100 [76] GPUs for ImageNet training and a single TITAN Xp GPU [75] for CIFAR training. I build PruneTrain using PyTorch [79]. Because of limited resources, I perform sensitivity evaluation primarily with CIFAR and evaluate functionality and final efficiency with ImageNet.

Table 4.1: Training FLOPs and time compared to the dense baseline: top1 validation accuracy of the dense baselines for CIFAR10: ResNet32 (93.6), ResNet50 (94.2), VGG11 (92.1), VGG13 (93.9), and for CIFAR100: ResNet32 (71.0), ResNet50 (73.1), VGG11 (70.6), VGG13 (74.1), and for ImageNet: ResNet50 (76.2)

Dataset	Model	Val. Accuracy Δ (fine-tunning)	Train. FLOPs (time)	Inf. FLOPs
CIFAR10	ResNet32	-1.8%	47% (81%)	34%
	ResNet50	-1.1%	50% (81%)	30%
	VGG11	-0.7%	43% (57%)	35%
	VGG13	-0.6%	44% (57%)	37%
CIFAR100	ResNet32	-1.4%	68% (88%)	54%
	ResNet50	-0.7%	47% (66%)	31%
	VGG11	-1.3%	53% (74%)	43%
	VGG13	-1.1%	58% (67%)	48%
ImageNet	ResNet50	-1.87% (-1.58%)	60% (71%, *66%)	47%
		-1.47% (-1.16%)	70% (76%, *72%)	56%
		-0.24% (+0.20%)	97% (98%, *98%)	88%

* Measured using V100 GPUs

4.4.2 Model Pruning and Training Acceleration

I first present the evaluation results on CIFAR and ImageNet in Table 4.1. I report 4 metrics: the training and inference FLOPs (FP operations), measured training time, and validation accuracy. Training time does not include network architecture reconfiguration time, which I do optimize and occurs only once in many epochs. I use 182 epochs [36] and 90 epochs to train CNNs on CIFAR and ImageNet, respectively.

For ResNet32 and ResNet50 on CIFAR10, PruneTrain reduces the training FLOPs by ∼50% with a minor accuracy drop compared to the dense baseline. The compressed models after training show only 34% and 30% of the dense baseline inference cost for ResNet32 and ResNet50, respectively. The results of ResNet32/50 on CIFAR100 show similar patterns, which exhibits the robustness of PruneTrain, given that CIFAR100 is a more difficult classification problem. For CIFAR100, PruneTrain reduces the training and inference FLOPs by 32% and 46% for ResNet32, and 53% and 69% for ResNet50, while losing only 1.4% and 0.7% of validation accuracy, respectively compared to the dense baseline. These results show that PruneTrain reduces more training FLOPs from a deeper CNN model, since more unimportant channels and layers are sparsified and removed early in the training. PruneTrain also achieves high model compression with similar validation accuracy loss for both VGG models on CIFAR.

PruneTrain also shows high training cost savings for ResNet50 trained on ImageNet: 40%, 30%, and 3% for three different pruning strengths (0.25, 0.2, and 0.1). Thus, I conclude that PruneTrain is robust to changes in CNN model and dataset complexity. The trained ResNet50 shows 53%, 44%, and 12% reduced inference FLOPs with 1.87%, 1.47%, and 0.24% accuracy loss, respectively. In addition, with extra training epochs for fine-tuning without group lasso regularization, I could recover 0.3% additional accuracy for the regularization strengths of 0.25 and 0.2, and achieve even better accuracy than the baseline by 0.2% for the regularization strength of 0.1. Although not shown in the table, PruneTrain also saves 37%, 33%, and 5% of off-chip memory accesses of BN (batch normalization) layers for ResNet50 with the

Table 4.2: Inference performance comparison (number of images per second and relative speedup by PruneTrain). The three ResNet50 results on ImageNet use different regularization strengths of 0.25, 0.2, and 0.1.

Dataset	Model	Batch size=10		Batch size=100	
		Base	PruneTrain	Base	PruneTrain
CIFAR100	ResNet32	3038	4081 (1.34×)	18587	24759 (1.33×)
	ResNet50	1442	1442 (1.18×)	7847	11865 (1.51×)
	VGG11	5534	5534 (1.44×)	15489	23878 (1.54×)
	VGG13	5197	5197 (1.38×)	12845	21075 (1.64×)
ImageNet	ResNet50	610	937 (1.53×)	772	1194 (1.55×)
			833 (1.36×)		1047 (1.36×)
			661 (1.08×)		813 (1.05×)

three different regularization strengths. Since the performance of BN layers is bounded by memory access bandwidth, reducing their memory traffic has a significant impact on the overall CNN model training time.

The measured training time reduction is smaller compared to the saved training FLOPs across datasets and CNN models. This is mainly caused by the reduced data parallelism at each layer after pruning, which decreases GPU execution resource utilization. Also, SIMD utilization within the GPU cores decreases for some layers due to the irregular channel dimensions after pruning and reconfiguration. In particular, for CIFAR10 and and CIFAR100, ResNets shows lower training time saving compared to VGGs, because it has many layers with reduced parallelism. In comparison, VGG has fewer layers with wider data parallelism and utilization is impacted less by pruning. For ImageNet, the training time saving of ResNet50 is bigger when V100 GPUs are used. This is because high off-chip memory bandwidth of V100 [50] makes

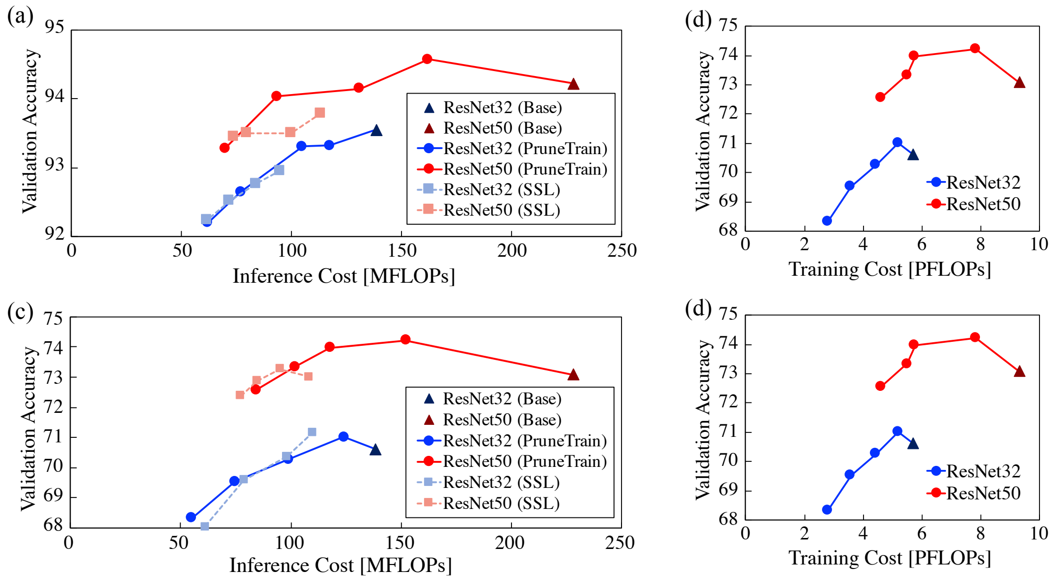


Figure 4.8: (a) Inference FLOPs and the validation accuracy by different regularization ratios of PruneTrain and SSL for ResNet32/50 on CIFAR10 (c) and on CIFAR100, (b) Training FLOPs and BN cost by accuracy of PruneTrain for ResNet32/50 on CIFAR10 and on (d) CIFAR100. (The triangles in all figures represent the dense baseline)

the execution time portion of memory bandwidth-bound layers smaller, which eventually makes the training time saving by the pruned computations more visible in the overall training time.

I also compare the performance of the trained models in terms image inferences per second (Table 4.2). I evaluate using two different batch sizes of 10 and 100 [71], on one TITAN Xp GPU. Overall speedup of PruneTrain is lower than the saved inference FLOPs in Table 4.1 because of resource underutilization. Therefore, processing 100 images shows performance that is equal to, or slightly better than the batch size of 10. Also, since ResNet50 for ImageNet has more channels, its PruneTrain inference performance is better than the CNN models for CIFAR100 given the ratio of their pruned FLOPs.

4.4.3 Comparison to Prior Work

4.4.3.1 Comparison to Pruning From a Pre-trained Model

I verify that pruning while training from scratch shows comparable compression quality and accuracy as following the current best practice of training from a pre-trained model as done by SSL [103]. The comparison results are summarized in Figure 4.8, which plots the tradeoffs between both inference and training cost and validation accuracy for ResNet32/50 on CIFAR10/100. I sweep the group lasso penalty ratio from 0.05 to 0.2 with an interval of 0.05. Since Wen et al. [103] do not discuss how to set the group lasso penalty coefficient, I apply my proposed mechanism to SSL as well.

Results for inference (Figure 4.8a and Figure 4.8c) demonstrate that PruneTrain is, in fact, superior to pruning from a pre-trained model. I make three important observations. First, for ResNet50, PruneTrain attains higher accuracy than the baseline dense model while still reducing cost. Accuracy is highest at around 150 MFLOPs/inference compared to the dense 230 MFLOPs/inference. I attribute this to the regularizer I use for pruning also leading to better generalization [110]. Second, PruneTrain and SSL achieve comparable accuracy-cost tradeoffs, yet PruneTrain offers a wider tradeoff range. Third, pruning is a very effective way to learn a good CNN model—starting from the complex ResNet50, PruneTrain is able to learn a network model that is simultaneously more accurate and lower-cost to use.

Figure 4.8b and Figure 4.8d show the training-cost tradeoff curve. I do not show the training cost of SSL, because its training protocol first trains the dense network and then prunes, resulting in a cost that's almost 3 times higher than baseline. PruneTrain reduces computation cost with a minor accuracy loss compared to the dense baseline (triangles in the graph). The shape of computation tradeoff curve is similar to that of inference. Because PruneTrain gradually and continuously prunes the network to reduce its training cost over time, it can start from the complex ResNet50 and learn a better model in less training time compared to conventional dense ResNet32 training. Although not shown in the graph, I also measure the memory traffic reduction in batch normalization layers. PruneTrain removes their memory traffic by similar accuracy tradeoff as the pruned output channels do not need normalization.

4.4.3.2 Comparison to Trial-and-Error Based Pruning

I compare the training results of PruneTrain to AMC (Auto ML for model compression) [38] to show that learning the architecture by regular-

Table 4.3: Comparison to AMC (Auto ML for Model Compression): compression results of ResNet56 on CIFAR10. The results of AMC are taken directly from [38].

Method	Base Val. accuracy	Validation accuracy Δ	Inference FLOPs	Removed layers
PruneTrain	94.5%	-0.5%	34%	18 (21%)
AMC	92.8%	-0.9%	50%	Not known

ization during training leads to a better compression and accuracy tradeoff than trial-and-error based pruning from a pre-trained model (Table 4.3). I use ResNet56 on CIFAR10 for comparison, which is the experimental setting used in AMC. While AMC reduces the inference FLOPs to 50% with 0.9% accuracy drop (after fine-tuning), PruneTrain reduces an additional 16% FLOPs while achieving higher accuracy by 0.4%. While the capability of learning network depth was not discussed in AMC, PruneTrain also learns depth and removes 21% of the convolution layers of ResNet56. This layer removal is effective in reducing the actual inference latency because pruning layers does not decrease data parallelism and does not affect compute-resource utilization.

4.4.4 Optimization and Sensitivity Evaluation

4.4.4.1 Dynamic Mini-Batch Size Adjustment

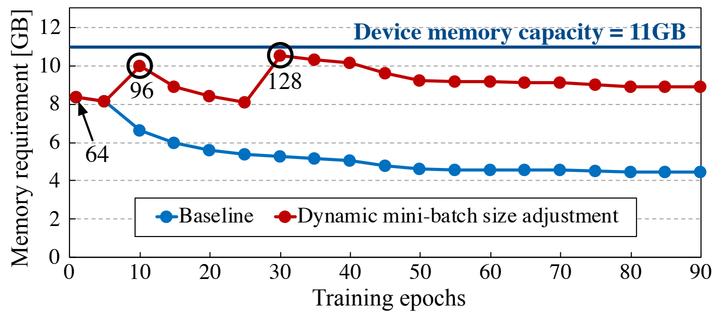
Figure 4.9 shows the off-chip memory requirement per GPU for a single training iteration using PruneTrain. I train ResNet50 for CIFAR100 and ImageNet datasets on a GPU with an 11 GB memory capacity (NVIDIA 1080 Ti). As training proceeds, the memory requirement gradually decreases due to pruning.

Once enough space is freed up, my proposed *dynamic mini-batch size adjustment* mechanism increases the mini-batch size to fully utilize the off-chip memory capacity. As shown in Figure 4.9a, for ImageNet, I start with a per-GPU mini-batch of 64 (total of 256 across 4 GPUs), which is the largest mini-batch that can fit in the off-chip device memory. As the memory requirement gradually decreases by pruning, I increase the per-GPU mini-batch from 64 to 96 and later to 128 at 10^{th} and 30^{th} epoch, respectively. The training context still fits in the GPU memory at each epoch. In this example, I use a mini-batch size adjustment granularity of 32 samples per GPU, but a smaller granularity can also be used.

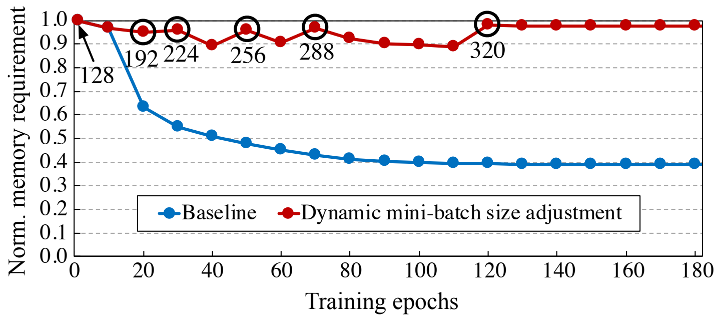
The memory required by ResNet50 for CIFAR100 is already small. Hence, in order to demonstrate the effect of dynamic mini-batch size adjustment in this case, instead of trying to fit the largest mini-batch size possible in the GPU memory, I start with the standard mini-batch size of 128 (Figure 4.9b).

Then, as PruneTrain gradually reduces the memory requirement, I gradually increase the mini-batch size such that I maintain similar device memory capacity utilization. This is shown in Figure 4.9b, where I increase the mini-batch size by multiples of 32 up to, eventually, a mini-batch of 320, which is 2.5X larger than the initial mini-batch size. Note that increasing the mini-batch size not only increases the computational parallelism, it also linearly decreases the model update frequency. Reducing model update frequency can significantly accelerate distributed training by lowering inter-device communication and off-chip memory accesses.

Table 4.4 compares the training time reduction with and without dynamic mini-batch size adjustment. The table also compares the validation accuracy and final inference computation complexity in the two scenarios. While dynamic mini-batch size adjustment barely affects the quality of learn-



(a) ResNet50 on ImageNet: Memory requirement at every 5 epochs. The baseline mini-batch size per GPU of 64 is increased to 96 and 128 at the 10th and 30th epochs respectively, which fits the device memory capacity.



(b) ResNet50 on CIFAR100: Normalized memory requirement every 10 epochs. The baseline mini-batch size of 128 is increased to 192, 224, 256, 288, and 320 at the 20th, 30th, 50th, 70th, and 120th epochs respectively.

Figure 4.9: Memory requirement of one training iteration per accelerator during training epochs.

ing and pruning, it has a high impact on training time. Dynamic mini-batch size adjustment improves accuracy by 0.3% and raises the inference FLOPs by 3% for CIFAR100 and reduces accuracy by 0.04% and decreases the inference FLOPs by 1% for ImageNet. It reduces the training time by 57% and 34% (39% on a V100 GPU) compared to the dense baseline for CIFAR100 and

ImageNet, respectively. This is also an improvement of 26% and 17% (14% on a V100 GPU) compared to the naive PruneTrain for CIFAR100 and ImageNet, respectively. Although the training time is substantially reduced, the impact is less than expected, given that I are enabling 2× more computational parallelism with fewer model updates than the naive PruneTrain. I suspect that this is caused by a sub-optimal GPU convolution kernel choice that comes from the increased data parallelism only in mini-batch dimension.

Table 4.4: Training time, inference FLOPs, and validation accuracy of ResNet50 with and without dynamic mini-batch size adjustment. Top-1 validation accuracy of the dense baselines: ResNet50 trained on CIFAR100 (73.1) and on ImageNet (76.2).

Dataset	Model	Method	Train time reduction	Inference FLOPs	Val. Acc. Δ
CIFAR100	ResNet50	Naive	34%	31%	-0.7%
		Adjusted	43%	34%	-0.4%
ImageNet	ResNet50	Naive	29% (*34%)	47.4%	-1.87%
		Adjusted	34% (*39%)	46.4%	-1.91%

* Measured using V100 GPUs

4.4.4.2 Network Reconfiguration Interval

PruneTrain adds two hyper-parameters on top of dense training: sparsification strength, which I already discussed, and the reconfiguration interval. The reconfiguration interval affects training time by trading off the time overhead of manipulating the network model with greater savings of more-frequent pruning (actual removal of computation). The reconfiguration interval may also affect the compression and accuracy of the final learned model. Fortunately, the compression and accuracy achieved are insensitive to this hyperparameter, as shown in Figure 4.10, which shows the accuracy vs. computation cost tradeoff curve for different intervals. Thus, the interval can be chosen to balance per-iteration performance gains with reconfiguration time overhead. The overhead depends on the specific framework used. I find that reconfiguring a network architecture every 10 epochs for CIFAR or 5 epochs for ImageNet has small overhead in my experiments.

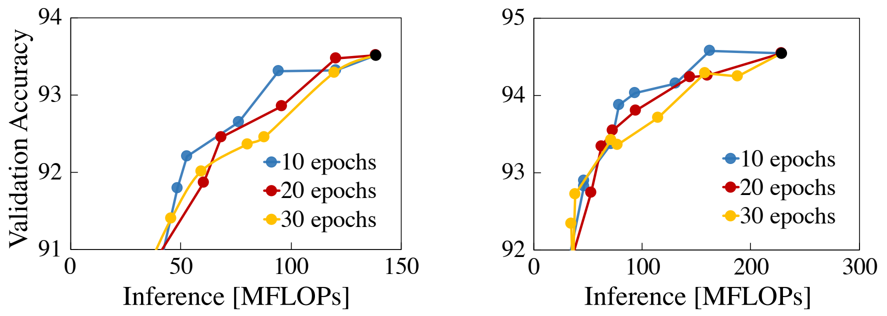


Figure 4.10: Reduced inference FLOPs and validation accuracy by different network reconfiguration intervals. ResNet32 (Left) ResNet50 (Right) on CIFAR10.

4.4.4.3 Communication Cost Savings in Distributed Training

4.4.4.4 Communication Cost Savings in Distributed Training

As training proceeds, the model size reduction by PruneTrain leads to decreasing communication cost between GPUs. Figure 4.11 shows the projected decrease in communication cost during the training of ResNet50 for ImageNet. I model the communication cost using ring allreduce. The figure shows the communication cost per training epoch normalized to the dense baseline for different sparsification strengths (therefore, different pruning rates).

Each time the network is reconfigured, the number of weights decreases, leading to a reduction in weight gradients communicated per training iteration. Furthermore, an aggressive sparsification strength (0.2 and 0.25) allows dynamic mini-batch adjustment to increase the mini-batch sizes (dotted lines), leading to further reduction in communication cost for later epochs. Overall, PruneTrian saves 55% average communication cost regardless of the number of GPUs used for distributed training. This pruning-based communication reduction is orthogonal to other existing techniques for communication reduction in distributed training, e.g. weight gradient compression and efficient gradient reduction mechanisms, which can be used in conjunction with PruneTrain for further communication improvements.

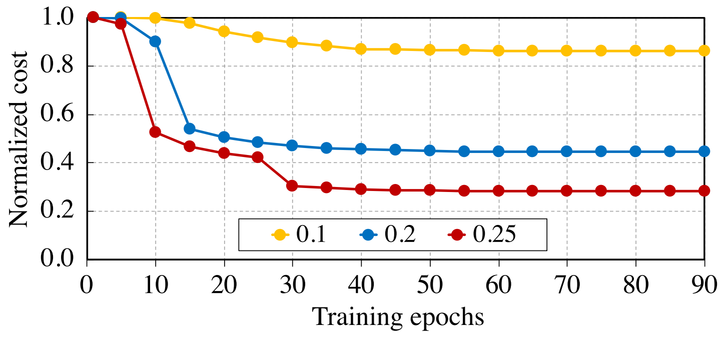


Figure 4.11: Projected per-epoch communication cost of model updates based on hierarchical ring-allreduce. The communication cost is normalized to the cost of dense baseline ResNet50 training on ImageNet for different group lasso regularization penalty ratios.

4.4.4.5 Individual Weight Sparsity

PruneTrain uses structured pruning of channels (and possibly layers) to learn a smaller, yet still dense model. This is important for high-performance execution on current hardware. However, the regularization leads to weight sparsity even within the remaining channels. Figure 4.12 shows the density of channels (input channel density × output channel density) and the density of weights for each layer in ResNet50 trained on ImageNet. Roughly half of all weights within the remaining channels (roughly half of all dense channels) are also near-zero and can be pruned. Such unstructured sparsity can be utilized to store the pruned model in a compressed form and to possibly further speed up execution if the inference hardware supports efficient sparse computations [32].

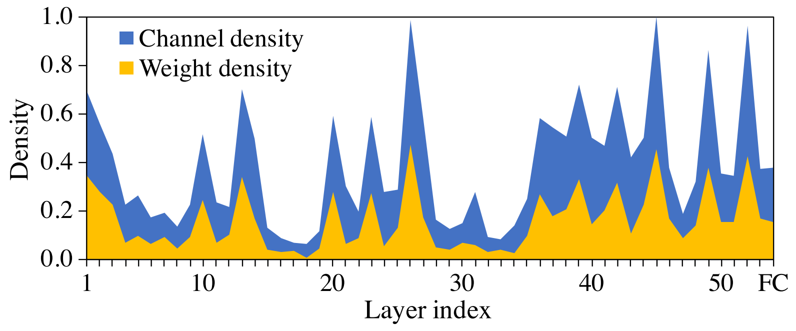


Figure 4.12: Channel and weight density of each layer. (ResNet50 trained on ImageNet using PruneTrain) The number in the x-axis indicates the convolution layer index.

4.5 Discussion

4.5.1 PruneTrain with Mini-batch Serialization

Provided that MBS (Mini-batch Serialization) saves significant off-chip memory traffic between network layers as discussed in Chapter 3, it is natural to ask if PruneTrain can be used with MBS. Unfortunately, MBS cannot be used along with PruneTrain since they have a conflict on the choice of the feature normalization algorithm.

PruneTrain uses BN (Batch Normalization) for feature normalization, which is crucial for channel-wise structured pruning. BN normalizes the features at the granularity of channels. Thus, pruning a channel does not affect the features in other channels, making continuous training after channel pruning algorithmically valid. Also, because the feature normalization granularity matches with the pruning granularity, once a channel is sparsified, the output of a normalization layer followed by affine transformation remains near-zero, making structured sparsification easier across layers.

Unlike PruneTrain, MBS relies on GN (Group Normalization) to partially serialize a mini-batch of samples. Therefore, if channel-wise pruning is used with MBS, each network structure reconfiguration breaks the algorithmic hyperparameter setting, the number of channels per normalization group. Also, because the granularities of pruning and feature normalization are different, the convolution layer outputs from a sparsified channel become nonzero after the normalization with the outputs from other channels. I trained ResNet50 with both PruneTrain and MBS and it shows worse results in both accuracy and pruning rate compared to PruneTrain without MBS.

4.6 Summary

In this chapter, I propose PruneTrain, a mechanism to accelerate the network model training from scratch, while the eventually pruned model enables fast inference. PruneTrain relies on structural pruning using group lasso regularization. It continuously sparsifies parameters using regularization and reconfigures the network architecture into a small and dense form during training, so as to take advantage of the reduced model size not just during inference, but also during training. This is based on my observation that while pruning with group lasso regularization, once a group of model parameters are forced to near-zero magnitude, they rarely revive during the rest of the training.

I propose three key optimizations for efficient implementation of PruneTrain. First, I update the group lasso regularization penalty coefficient such that I enable achieving high model pruning rate with minor accuracy loss during a single training run from scratch. Second, I introduce channel union, a way to prune CNN models with short-cut connections to lower the overheads from naive channel indexing and tensor reshaping. Lastly, I dynamically increase the mini-batch size while training with PruneTrain, which increases the data parallelism and reduces the frequency of model updates that involves off-chip memory accesses and inter-GPU communication, leading to further training time saving. Altogether, PruneTrain cuts the computation cost of training modern CNNs (represented as ResNet50) at least by half, and up to 53% and 40% for small and large datasets, enabling 34% and 39% reduction in end-to-end training time respectively.

Chapter 5

Flexible Systolic Array Architecture for Fast and Efficient Pruned Model Training

This chapter discusses an accelerator architecture and layer workload scheduling for fast and energy-efficient training of pruned network models. I first analyze the trade-offs of training accelerator designs with different sized GEMM execution cores, such as WaveCore, which uses a single large systolic array core and a potential accelerator design with many small such cores. Then, I show that using a large core suffers low PE utilization when processing pruned CNNs. Although many-small-core designs achieve high PE utilization, they decrease on-chip input reuse and incurs area overhead. To support PE utilization and input reuse in processing both dense and pruned DNN models, I propose a flexible systolic array architecture. I also propose a coupled GEMM instruction selection heuristics for efficient utilization of the flexible hardware resources. Finally, I show the gains of the proposed accelerator in performance and energy efficiency.

5.1 Challenges of Training Pruned CNN Models

Channel pruning is a commonly used technique to make a CNN model fast and energy-efficient for network inference [103, 102, 25, 4, 113, 72, 38]. Channel pruning is done by retraining a pre-trained dense model or pruning unimportant channels during training. In both cases, training accelerators have to deal with both dense and pruned models. Unlike unpruned models that have regular number of channels, such as 64, 128, and 256 (powers of two) [57, 85, 84, 101], their channel-pruned versions have a reduced number of channels (e.g., 3, 71). These convolution layers are computed as GEMMs and the GEMM dimensions of pruned models also have severely reduced sizes. Tiling and mapping such GEMMs to the large *GEMM cores* (GEMM execution cores, typically systolic arrays) of a modern training accelerator is inefficient: Many GEMM tiles tend to have sizes smaller than the size of the GEMM core (e.g., 128×128) and pruned such GEMM tiles can not fully utilize the PEs of the core. Therefore, although a CNN requires fewer FLOPs than the unpruned baseline, actual speedup on this reduced workload depends on the efficiency of mapping the GEMMs of (partially) pruned layer to large GEMM cores.

5.1.1 Trade-offs of Different GEMM Core Designs

CNN training accelerators are typically designed to train network models with high data parallelism in all tensor dimensions (e.g., channels, mini-batch, feature sizes). For efficient execution of such the large GEMMs resulting from this high parallelism, many modern training accelerators adopt large systolic-array cores. This design principle is also chosen by WaveCore and many other training accelerators, e.g., Google's TPU [21] and Intel's Nervana NNP.

However, convolution and FC (fully-connected) layers in a pruned CNN model exhibit reduced and irregular GEMM dimensions. When one GEMM dimension becomes smaller than the height or width of a core, PEs in the core are not fully occupied and this causes severe resource underutilization and slowdown. Figure 5.1 shows the PE utilization of WaveCore when executing training iterations of both dense and pruned ResNet50 models [36]. This result is estimated using instruction-level simulator used to evaluate Mini-batch Serialization (Chapter 3). The baseline ResNet50 is pruned using PruneTrain, a channel pruning algorithm using group lasso regularization (Chapter 4). In

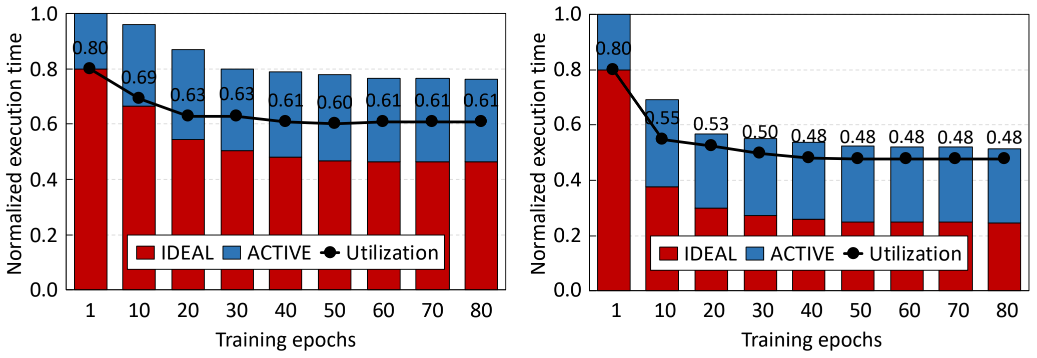


Figure 5.1: Execution time of channel-pruned ResNet50 normalized to the execution time of the dense baseline (the left-most bar in each figure). The weights are regularized and pruned using different pruning strengths: (left) medium strength and (right) high strength. Only the execution time of convolution and FC layers is shown considering that they account for >98% of the FLOPs of each training iteration.

this experiment, I use a pruning interval of 10 epochs and the two figures show the results of using different pruning strengths: medium pruning strength that removes restricted number of channels with small accuracy loss (left) and high pruning strength that removes more channels with larger accuracy loss (right). Each of the pruned convolution and FC layers is converted into a single GEMM using im2col. A WaveCore with a single 128×128 systolic array executes each blocked GEMM tile using input-stationary dataflow (explained in Section 3.3). The figure shows two bar graphs: the red bars (*IDEAL*) indicate the execution time with the assumption of 100% PE utilization and the stacked blue bars (*ACTIVE*) show the the additional execution time caused by PE underutilization. The PE underutilization is solely caused by the size mismatch between GEMM tiles and the core and is estimated by using infinite memory BW. As the red bars show, PruneTrain gradually reduces the FLOPs of the baseline models to 46% and 25% for the medium and high pruning strength, respectively. However, due to the large core size and reduced GEMM dimensions in many convolution layers, PEs are highly underutilized and exhibit overall PE utilization of only 65% and 55% for the medium and high pruning strengths respectively. Also, even the baseline model shows only 80% PE utilization and this is because some early convolution layers have small number of channels.

5.2 Naive GEMM Core Splitting

Diminishing PE Utilization and Increasing Input Load Cost. The PE underutilization caused by small GEMM dimensions can be mitigated by splitting a large GEMM core into multiple small cores. However, this many-small-core design can be inefficient in processing typically large GEMM tiles, which account for most GEMM tiles in both pruned and unpruned CNN layers. First, small cores have low in-core input reuse compared to a large core, which increases input traffic from memory. Second, having more cores increases area complexity of wires, data path switches, and SRAM buffer control and decoding logic.

To observe the relation between the core size and PE utilization, I conduct a simple experiment that estimates overall PE utilization and the volume of on-chip input traffic while training ResNet50 using PruneTrain. The evaluation is done with different core configurations (Figure 5.3). Again, I use ideal memory BW to show the PE underutilization caused by the size mismatch between GEMM tiles and the core. The baseline accelerator design used in this experiment has a group of GEMM cores sharing a global buffer (GBUF) (Figure 5.2.a). The GBUF is used to block GEMM inputs. This baseline design also has a pair of local buffers (LBUFs) in each GEMM core for input double buffering to hide input load latency. The blue lines in Figure 5.3 show PE utilizations and the red lines show average input traffic volume from the GBUF to LBUFs when using medium (plain lines) and high (dotted lines) pruning strengths. Both PE utilization and input traffic volume are averaged across training iterations over the whole training, thus they encapsulate the results of the baseline unpruned model and many different intermediate pruned models.

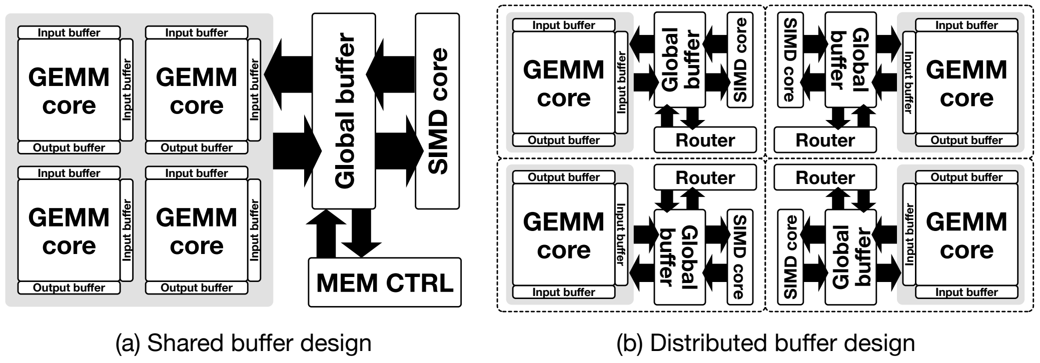


Figure 5.2: Different GBUF configurations.

Splitting one 128×128 core into four 64×64 improves PE utilization by as much as 23% but it also increases input load traffic by 1.7× due to reduced input reuse. Further splitting the cores yields diminishing PE utilization improvement but continues to increase input traffic. Using 16× (32×32) and 64× (16×16) cores increase PE utilization by only 8% and 4% but increases input load traffic by 3.4× and 6.6×. Also, 32×32 and 16×16 cores show almost similar PE utilization for the two models with different pruning strengths. This indicates that small cores do not necessarily further improve the PE utilization of the more aggressively pruned CNN models. *Overall, this result shows that reducing core size less than 32×32 is not cost-efficient.*

High Area Overhead of the Many-Small-Core Design. The increased input traffic of small cores requires higher on-chip data BW: when the core count increases by 4×, the on-chip data BW required increases by 2×. The impact of increasing input BW is different depending on the accelerator design.

Figure 5.2 shows two potential buffer designs. In design (a), cores share

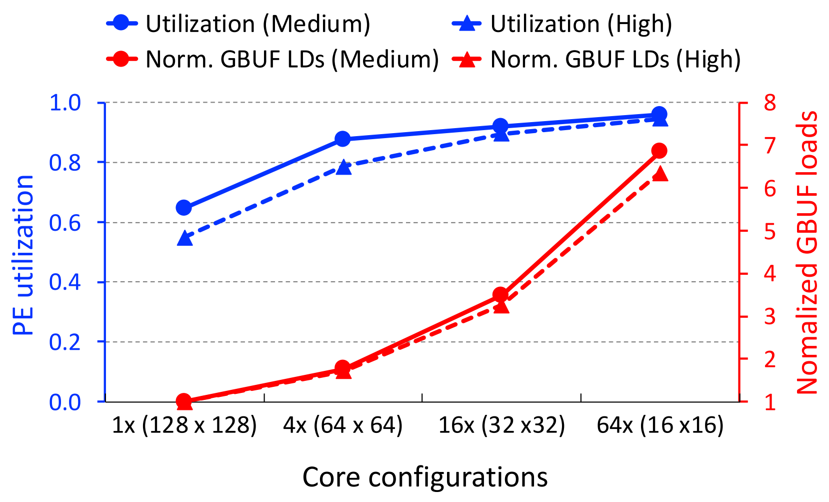


Figure 5.3: PE utilization and GBUF load traffic increases for pruned ResNet50 training by splitting a single large core into many small cores. Medium and High indicate channel pruning strength.

a GBUF, and design (b) there is a dedicated GBUF for each core. A shared buffer needs additional data paths from GBUF to the LBUF for each core, along with path switches. On the other hand, a distributed buffer design has less data path-associated area overhead but it involves the area overhead of splitting LBUFs and GBUFs (duplicating decoding and data repeating logic). In the distributed buffer design, the inputs should be allocated to cores carefully to avoid input replication. Although inter-core shared data can be split across cores, this leads to data transfers over potentially low-BW inter-core channels. A more general accelerator architecture mixes these two designs by creating multiple groups of cores such that cores in each group share a GBUF [28].

I compare the area overhead of different core configurations that have different cores sizes and buffer designs (Figure 5.4). For simple estimation, this experiment considers only the area of PEs, SRAM buffers, and data paths. I use CACTI 7.0 [12] to estimate the area overhead of buffers and use the module size of a mixed precision multiplier and adder units [111] to estimate the area of the systolic array. When the number of cores becomes is greater than four, I group the cores such that they share a GBUF and this configuration is shown in the second row of the X-axis (G and C indicate groups and cores, respectively). The blue line graph in Figure 5.4 shows the area overhead caused only by the additional logic, coming from splitting GBUFs and LBUFs and the red line indicates the area overhead for increased data paths.

The overhead of data paths is estimated conservatively assuming the wires do not overlap with logic (so the actual overhead should be smaller). The increase in chip width and height from additional data paths is estimated by distributing the wires to 5 metal layers using a wire pitch of 0.22um, similar to DaDianNao [17]. The overheads of different core configurations are normalized to the area of a single 128×128 core. Splitting a single core to 4 (64×64) cores has a relatively small area overhead of 4%, and this (❶) mainly comes from doubling the data paths by having cores share a GBUF. Further splitting to 16 (32×32) cores increases area overhead to as much as 13% and this overhead is caused by dividing a GBUF to four parts (❷). Finally, 64 (32×32) cores have a 23% area overhead. The additional area overhead comes from increasing the number of cores sharing a GBUF in each group. *Overall, this experiment shows that splitting a large core into ≥16× small cores is not an area-efficient* *design option.*

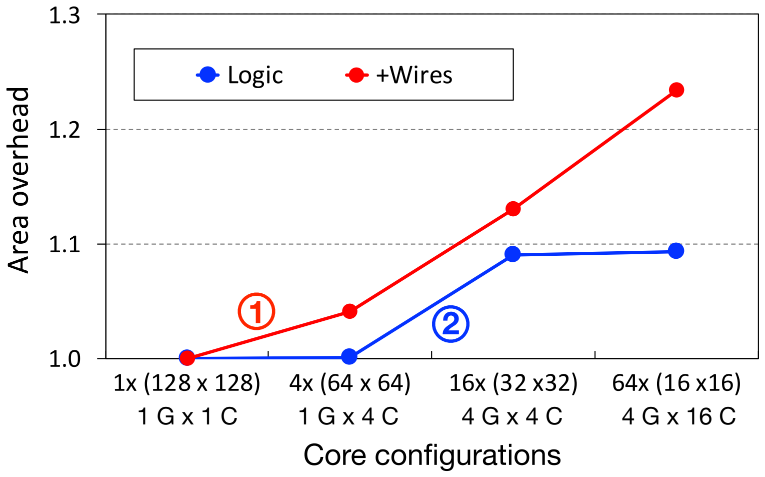


Figure 5.4: Area overhead of core division normalized to 1× (128×128).

In summary, naively splitting a large core into many small cores improves PE utilization but its gain diminishes as the splitting factor increases. In addition, the many-small-core design increases total traffic from the GBUF due to the reduced in-core input reuse and also requires high area overhead. To better balance the tradeoffs, *a new GEMM core architecture is needed to maintain high input reuse as using a large core and achieve high mapping flexibility as when using multiple small cores.*

5.3 FlexSA: Flexible Systolic Array for High PE Utilization and Input Reuse

In this section, I introduce a flexible systolic array architecture for efficient processing of slim and fat GEMMs with irregular shapes. Then, I present a compile-time GEMM tiling and scheduling methods for efficient utilization of the proposed core designs.

5.3.1 FlexSA Core Architecture

To map small GEMM tiles to cores with high PE utilization, splitting a large core into smaller cores is unavoidable. However, processing a large GEMM tile using many small cores is inefficient as this increases input traffic from GBUFs. This is a critical problem as GBUF access cost becomes expensive as the size of a GEMM core becomes bigger. To take advantage of the benefits of both a large core and multiple small cores, my idea is to use a group of cores that collaborate when processing a large GEMM tile and work independently when processing small GEMM tiles. To this end, I propose **FlexSA**, a flexible systolic array architecture that meets the need for both high input reuse and PE utilization.

Figure 5.5 shows the logical structure of FlexSA. FlexSA is based on four systolic array cores sharing a GBUF, as illustrated in Figure 5.2.a. However, unlike the baseline design, where each core operates independently, FlexSA can reconfigure its operation mode such that the four cores work independently or collaboratively. FlexSA provides four different operating modes and they are supported using additional data paths and path switches (colored red in Figure 5.5).

Sub-Array Operations. The four different systolic operating modes are illustrated in Figure 5.6. **FW** (full wave) uses the four cores as a single systolic

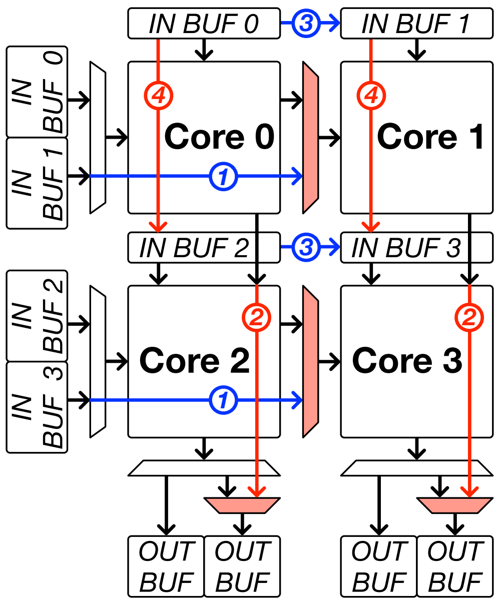


Figure 5.5: FlexSA architecture.

array by sharing inputs and passing partially accumulated outputs between cores (Figure 5.6.a). First, the inputs, vertically shifted from the LBUF on top of each core, are all unique and they are pre-loaded to each PE for the input-stationary systolic dataflow. Compared to a single large systolic array, this saves half the input shifts due to the reduced core height. Then, both cores 0 and 2 pass the inputs shifted from the left LBUFs to core 1 and 3 for reuse in MAC (multiplication and accumulation) operations (with the stationary inputs already pre-loaded in core 1 and 3). Also, cores 0 and 1 pass their outputs (partial sums) directly to core 2 and 3 for output reuse. Because the reuse of both vertically and horizontally shifted inputs is doubled, FW has significantly less input traffic compared to a naive independent four-core design.

FlexSA supports two systolic sub-array operations that use two pairs of cores independently. First, **VSW** (vertical sub-wave) forms two vertical systolic sub-arrays by paring the two cores in each column (cores 0 and 2 for one sub-array and cores 1 and 3 for the other as shows in Figure 5.6.b). This mode is designed for efficient processing of skinny GEMM tiles whose tile width is smaller than or equal to the width of one core. VSW starts by pre-loading the same inputs to each of the two vertical sub-arrays. To reduce the cost of sending identical inputs from GBUF to the LBUFs of both sub-arrays, I construct switchable data paths between the LBUFs of cores 0/2 and 1/3 for local broadcast ❸. After pre-loading the stationary inputs, the other inputs are horizontally shifted to each sub-array. To provide different inputs to each sub-array in parallel, VSW uses the additional horizontal data paths ❶. Also, VSW uses only a half of the output buffers, as shown in Figure 5.6. Therefore, it is possible to interleave two VSWs that use the same stationary inputs but different horizontally shifted inputs. Overall, VSW improves PE utilization by executing two skinny GEMM tiles and also improves stationary input reuse by 2× through local broadcasting.

Second, **HSW** (horizontal sub-wave) constructs a two horizontal systolic sub-arrays by paring the two cores in each row (cores 0/1 for one sub-array and cores 2/3 for the other as shows in Figure 5.6.c). HSW is designed for efficient mapping of fat GEMM tiles whose accumulation depth is smaller than or equal to the height of one core. HSW first pre-loads the same stationary inputs to each horizontal sub-arrays. Then, the other inputs are shifted from the left LBUFs through both paired cores in each row. Similar to the local input sharing used in VSW, I construct a direct data path between the input buffers of cores 0/1 and cores 2/3 to avoid duplicated input transfers from the GBUF ❹. The outputs from cores 0 and 1 are directly stored and accumulated at the output buffers at the bottom of cores 2 and 3 ❷. Overall, HSW improves PE utilization by processing two fat GEMM tiles in parallel and doubles the reuse of horizontally shifted data compared to using small cores.

Lastly, FlexSA supports full independent core operation by providing unique inputs to each core as shows in Figure 5.6.d, which I call **ISW** (independent sub-wave). This mode helps maintain high PE utilization when both GEMM tile width and accumulation depth are small. ISW provides independent inputs horizontally to cores 1 and 3 using the additional data path ❶ and the outputs of core 1 and 2 are send to and accumulated at the output buffers using the added vertical data path ❷. Compared to the independent core design, ISW exhibits lower input loading cost because FelxSA locally broadcasts the stationary inputs between cores. However, since its input reuse is the lowest among all modes, other sub-array modes should be prioritized over ISW for cost-efficient model training.

Area Overhead Estimation. FlexSA requires additional area for logic and data path wires and switches. I estimate area in 32nm technology and I compare the overhead relative to the naive four-core design. Again, I conservatively estimate the area overhead of additional data paths assuming they do

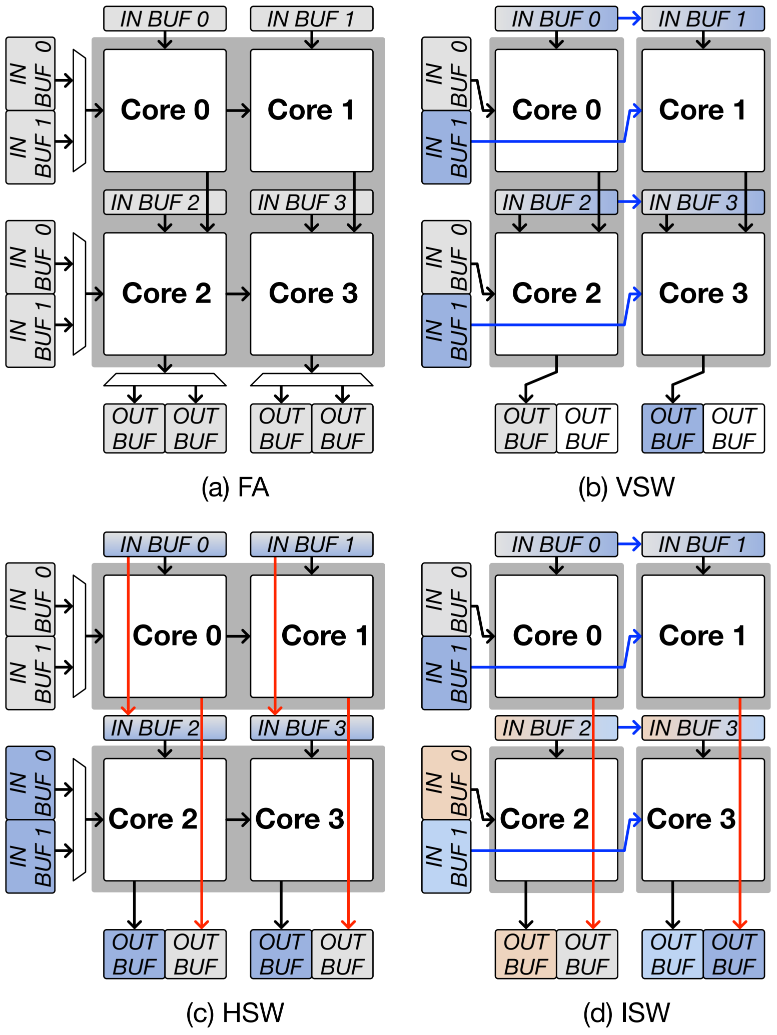


Figure 5.6: Four different systolic sub-array operations supported by FlexSA and the micro-architecture settings for each mode.

not overlap with logic. First, the addition of data path switches (1:2 MUXs) that select inputs and partial sums increases logic area by only $0.03mm^2$. Next, FlexSA also requires each PE at the top row of cores 2 and 3 to support a mixed-precision FMA (fused multiplier and adder) instead of just a 16-bit multiplier to (needed to accumulate the partial sums shifted from cores 0 and 1). This change increases the area by $0.32mm^2$. Also, the repeaters to drive signals over a core add $0.25mm^2$, where I use fanout of 32. The vertical wires connecting the outputs of cores 0 and 1 to the output buffers ❷ expand the width of the core by $0.09mm$.

However, the other wires do not affect die size because they are effectively canceled by the wiring overhead of connecting GBUFs and the LBUFs in the baseline four-core design. Overall, FlexSA increases the die area over the naive four-core design by only 1%. Again, this area estimate is conservative and placing the newly added vertical wires over PE array (as illustrated) can effectively hide the wiring area overhead. This is feasible because PE arrays use only a few low-level metal wires for local routing.

5.3.2 FlexSA Compilation and GEMM Tiling

FlexSA mode selection is crucial for efficient GEMM processing. Modes should be selected to achieve the highest PE utilization with minimum on-chip data traffic. Mode selection is coupled with GEMM tiling because different modes are optimal for GEMM tiles with different shapes. In this section, I introduce a compile-time GEMM tiling heuristics for tiling a GEMM into waves [1] to best utilize FlexSA resources for both high performance and energy efficiency.

[1]a wave is the GEMM execution granularity using the systolic array, which is defined in Section 3.3

The GEMM waves are mapped to cores and executed using different FlexSA modes. Different modes require different micro-controls, such as data path selection, input shifting, output shifting, and partial sum accumulation. For efficient communication between software and the FlexSA microarchitecture, I introduce a set of instructions that define the micro-controls needed for each FlexSA mode and for handling data transfers between on-chip buffers.

5.3.2.1 GEMM Tiling Heuristics

The proposed compile-time GEMM tiling heuristics choose waves that execute on FlexSA with the highest PE utilization and in-core input reuse. In other words, GEMM waves that uses the FlexSA modes with greater input reuse are prioritized and the GEMM waves using the FlexSA modes with lower input reuse are chosen only when using them improves PE utilization (i.e., FW>HSW=VSW>ISW).

GEMM Tiling Conditions for FW. The heuristics tile the large GEMM into multiple waves for processing on the systolic array. The tiling factors (blk_M, blk_N, and blk_K in Figure 5.7.a) are chosen to match the size of a full FlexSA core to maximize reuse by utilizing the FW mode. The size of blk_N and blk_K are equal to the width and height of a full FlexSA core, both of which are 128 in my baseline design. The size of blk_M is the size of the LBUF for non-stationary inputs divided by the height of a full FlexSA core, which is 256 using a 64KB LBUF. Because GEMM dimensions are not always a multiple of the ideal tile size, the execution of some tiles with FW would degrade utilization and performance. Such tiles (typically just the edge tiles of a large GEMM) are executed with the other FlexSA modes, again attempting to maximize utilization and reuse by prioritizing the use of HSW and VSW over ISW as shown in Figure 5.7 and discussed below.

GEMM Tiling Conditions for HSW. However, when the *blk_K* size of a GEMM wave is small, using FW under-utilizes PEs and slows down the GEMM execution. Thus, when the size of *blk_K* is smaller than or equal to the height of one FlexSA core, the proposed heuristics execute the GEMM wave using HSW as shown in Figure 5.7.b. By executing a single fat wave as two independent waves using the two horizontal sub-systolic arrays, HSW increases PE utilization.

GEMM Tiling Conditions for VSW. In addition, the tiles at the GEMM

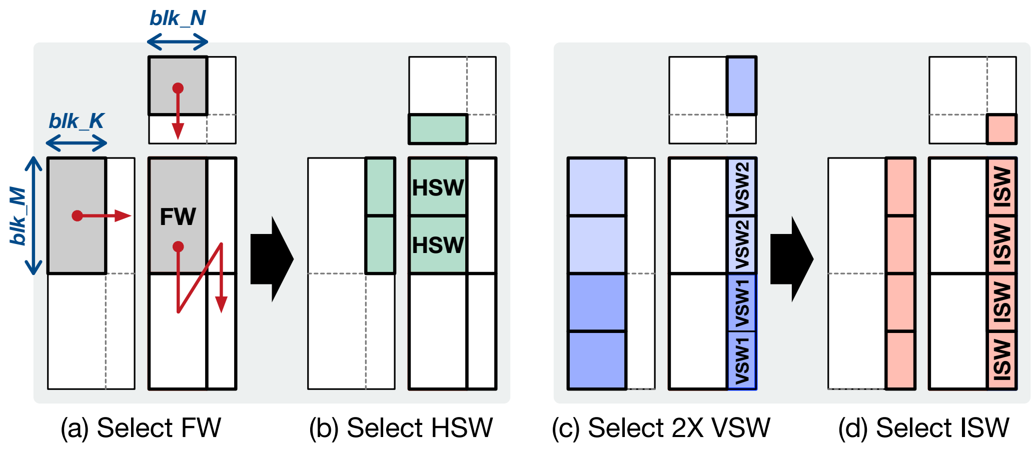


Figure 5.7: FlexSA operating mode selection examples.

edges can have small block sizes in the *blk_N* dimension and executing the GEMM waves of these tiles using FW also under-utilizes PEs. Therefore, when the *blk_N* is smaller than or equal to the width of FlexSA core, the heuristics execute the waves of the skinny GEMM tiles using VSW, which improves PE utilization by executing two waves in parallel using the two vertical sub-systolic arrays. This skinny GEMM tile is executed using two VSW operations (VSW1 and VSW2 in Figure 5.7.c) that can be interleaved, accumulating their results using half of the output buffers. Because the two VSW operations share the stationary inputs, this further improves input reuse.

GEMM Tiling Conditions for ISW. When both *blk_N* and *blk_K* are smaller than the width and the height of a FlexSA core, respectively, the proposed heuristics execute the GEMM waves using ISW. ISW executes a GEMM tile using four small independent waves, which operate in parallel to achieve high PE utilization (Figure 5.7.d). When a pair of VSWs precedes ISW, the four small GEMM waves executed by ISW accumulate their results with the partial sums stored in the output buffers. This combination achieves high PE utilization while maintaining high input reuse, minimally using ISW.

5.3.2.2 ISA Support and Compilation for GEMM Processing

To execute the GEMM waves using FlexSA, ISA support is needed. First, an instruction to define the configuration of each FlexSA modes is needed. This instruction provides the type of FlexSA mode and the sizes of each wave's *blk_M*, *blk_N*, and *blk_K*. This information is mapped to the micro-controls required to execute these GEMM waves such as data path switch selection, data shifting, and output accumulation. Second, vector load and store instructions are needed for data transfers between the GBUF (global buffer) and LBUFs (local buffers). Using vectors reduces instruction count and instruction decoding BW, compared to using word-granularity instruction. The compiler generates memory access routines that load inputs to the LBUFs of cores and store the GEMM outputs to the GBUF using these vector instructions.

Algorithm 1 GEMM execution flow

```
1: ▷ $blk\_M$, $blk\_N$, $blk\_K$: GEMM tiling factors
2: for $n \leftarrow 0$ to $N$ by $blk\_N$ do
3:     $n\_size = blk\_N$ if $(n+1) \times blk\_N < N$ else $N \pmod{blk\_N}$
4:     for $m \leftarrow 0$ to $M$ by $blk\_M$ do
5:         $m\_size = blk\_M$ if $(m+1) \times blk\_M < M$ else $M \pmod{blk\_M}$
6:         $wide\_wave$ = IsWideWave($n\_size$, $m\_size$)
7:         for $k \leftarrow 0$ to $K$ by $blk\_K$ do
8:             $k\_size = blk\_K$ if $(k+1) \times blk\_K < K$ else $K \pmod{blk\_K}$
9:             $tall\_wave$ = IsTallWave($k\_size$)
10:            ▷ Select FlexSA mode to execute the current GEMM wave
11:            $FlexSA\_mode$ = GetFlexSAMode($wide\_wave$, $tall\_wave$)
12:            ▷ Load stationary inputs to local buffers
13:            LdLBUF_V($GBUF\_ptr1$, $LBUF\_ptr1$, $k\_size, n\_size$)
14:            ▷ Shift stationary inputs to each PE
15:            ShiftV($k\_size, n\_size$)
16:            ▷ Load inputs to horizontally shift to a local buffer
17:            LdLBUF_H($GBUF\_ptr2$, $LBUF\_ptr2, k\_size, m\_size$)
18:            ▷ Execute a GEMM wave using systolic dataflow
19:            ExecGEMM($FlexSA\_mode$, $m\_size$, $n\_isze$, $k\_size$)
20:            __sync()
21:        ▷ Store GEMM outputs to a memory
22:        StLBUF($OBUF\_ptr$, $GBUF\_ptr3$)
```

The GEMM execution instruction generation algorithm of FlexSA compiler is shown in Algorithm. 1. The type of FlexSA mode is determined by the shape of the current GEMM wave. Before executing the wave, inputs are loaded from the GBUF to both LBUFs. For this, I use two vector load instructions **LdLBUF_V** and **LdLBUF_H**. These vector load instructions take address pointers for GBUF and LBUFs, and the size of the data to load. The LBUFs are double buffered to hide the access and transfer latency. Thus, the inputs to be used in the next GEMM wave execution iteration are pre-fetched in the current iteration.

Once stationary inputs are fully loaded at LBUFs, they are shifted to PEs using a **ShiftV** instruction. Using a separate instruction for stationary input shifting decouples it from the main GEMM wave execution step and makes it parallel to loading non-stationary inputs to the LBUFs, removing unnecessary execution step serialization. When both inputs are ready, an **ExecGEMM** instruction executes the target GEMM wave using the selected FlexSA mode. ExecGEMM takes four inputs: the type of FlexSA mode and the sizes of the GEMM wave dimensions (blk_M, blk_N, and blk_K). For HSW, VSW, and ISW, the sizes of these wave dimensions indicate those of each independent wave. When GEMM tile execution is complete after iterating over the K dimension, the outputs at the OBUFs (output buffers) are stored to the GBUF or to the off-chip memory using a **StLBUF** instruction.

5.4 Evaluation

5.4.1 Evaluation Methodology

I evaluate FlexSA using the instruction-level simulator that I also use to evaluate Mini-Batch Serialization Section 3.4. I prune ResNet50, the most commonly used CNN model for ImageNet. I also use Inception v4 [95], though only artificially pruning the model by applying the same pruning statistics of ResNet50. PruneTrain is used as the pruning mechanism, with two different model-pruning strengths. The model is trained for 90 epochs with a pruning interval of 10 epochs. The mini-batch size is 32. I use mixed-precision multiplication and accumulation [71] for the GEMMs of the convolution and FC layers.

I use five different accelerator configurations, as summarized in Table 5.1. The **1G1G** configuration uses a single 128×128 systolic array, **1G4G** splits this large core into four 64×64 cores, which share a single GBUF. The **4G4G** configuration further splits the cores into 16 32×32 cores, where I divide the cores and the GBUF into four groups with cores in each group sharing a GBUF. The **1G1FlexSA** configuration has a single 128×128 FlexSA with four 64×64 cores, and **4G1FlexSA** has four small FlexSA, each with a dedicated GBUF. I use a GBUF size of 10MB as used for Mini-Batch Serialization (Section 3.4) for all core configurations, and a single 270GB/s HBM2 for the memory system [50]. All local input and output buffers are sized to support double buffering. The local input buffers holding the horizontally shifted inputs are 2× larger than the input buffers holding stationary inputs for larger reuse of the pre-loaded stationary inputs.

5.4.1.1 GEMM Partitioning and Blocking.

The GEMMs in forward propagation and data gradient computation in back-propagation are skinny: they have large GEMM height (M), which is the product of the mini-batch and the feature size, and have small GEMM width (N), which is set by the number of channels. Thus when multiple core groups are used, I partition a GEMM across the M dimension with one partition per core group. On the other hand, the GEMMs for weight gradient computation in back-propagation have small GEMM dimensions in both height and width, but they have large accumulation dimension (K). In this case, I partition the GEMM across the accumulation dimension with one partition per group. Within each GEMM partition, I use 2-level GEMM blocking that holds the inputs of a multiple of GMEM tiles in the GBUF for reuse. Also, within each GEMM partition, each GEMM wave is allocated to cores in a group in round-robin fashion. When partitioning GEMM across the M dimension, different core groups use the same inputs from the GEMM N dimension. These shared inputs between core groups are replicated to avoid inter-group data transfers.

Table 5.1: Evaluation configuration description.

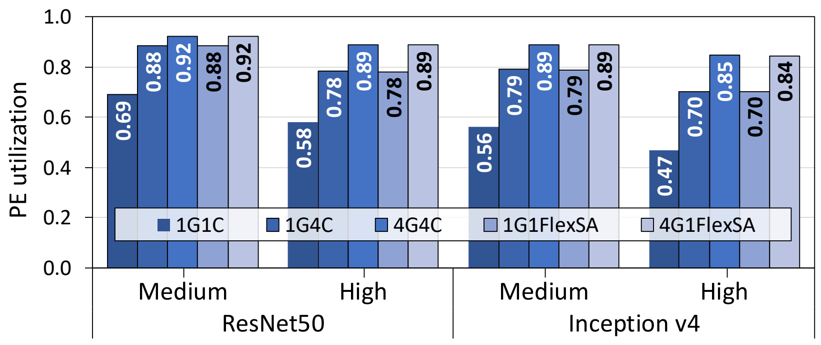
Configuration	Description
1G1C	1 group each with 1$\times$ (128$\times$128) core
1G4C	1 group each with 4$\times$ (64$\times$64) cores
4G4C	4 group each with 4$\times$ (32$\times$32) cores
1G1FlexSA	1 group each with 4$\times$ (64$\times$64) FlexSA
4G1FlexSA	4 group each with 4$\times$ (32$\times$32) FlexSA

I find that this method efficiently distribute GEMM workloads to multiple cores and core groups.

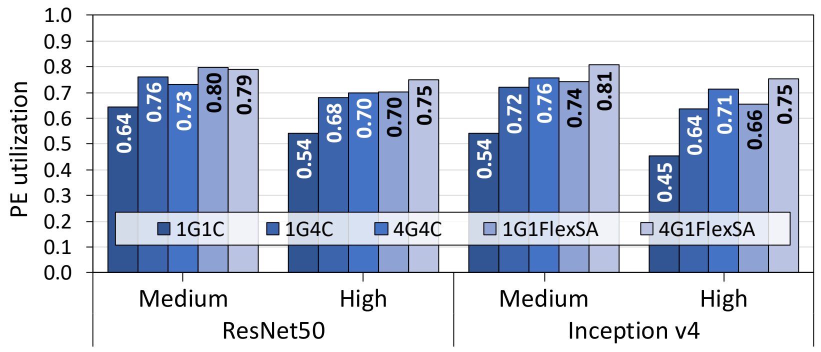
5.4.2 Evaluation Results

Figure 5.8a shows the average PE utilization of ResNet50 and Inception v4 using different core configurations. This experiment uses infinite DRAM BW to isolate the impact on PE utilization to only the mismatch between GEMM tile size and core size. Using a single large core (1G1C) shows low PE utilization for both CNNs. The PE utilization of Inception v4 is lower than that of ResNet50 because many of its convolution layers have fewer than 128 channels. Splitting the core into four small independent cores improves the PE utilization of both CNNs by $\geq$21% on average. Further splitting cores improves the PE utilization both CNNs as it mitigates internal PE utilization fragmentation but with diminishing returns. Using one FlexSA core (1G1FlexSA) and four small FlexSA cores (4G1FlexSA) exhibits only <0.1% ideal PE utilization compared to 1G4C and 4G4C, respectively. Considering that the PE utilization of using independent cores is the maximum that can be achieved with FlexSA. The result indicates that the FlexSA modes selected by the proposed heuristics achieve near-optimal PE utilization.

Figure 5.8b shows the impact on PE utilization with ResNet50 and Inception v4 using a single HBM2 for the DRAM system. Unlike the ideal memory BW case, this reflects the performance impact from GEMM input blocking, GEMM tiling, and GEMM scheduling. The 1G1C configuration



(a) Ideal DRAM BW



(b) 1× HBM2

Figure 5.8: PE utilization of different core configurations vs. pruning strength.

with HBM2 shows similar (slightly reduced) PE utilization compared to the results with ideal DRAM BW. This is because using a single 128×128 GEMM maximizes reuse requiring low peak memory BW. The many-small cores configurations (1G4C and 4G4C) exhibit greater PE utilization decrease when memory BW is considered. This PE utilization reduction is mainly caused by increased DRAM BW peaks from executing many small independent GEMM waves in parallel. Not all inputs of these small GEMM waves are coalesced thus the density of input loads is higher.

The two configurations using FlexSA show lower PE utilization drop compared to the naive many-core designs. This is because FlexSA uses small GEMM waves only when necessary. Furthermore, FlexSA offers two systolic array modes that use pairs of cores (VSW and HSW), which have higher in-core input reuse than using fully independent cores. This helps balance PE utilization and input reuse. Overall, 1G1FlexSA improves average PE utilization by 33% and 4% compared to the 1G1C and 1G4C designs, respectively. The 4G1FlexSA configuration improves average PE utilization by 7% compared to 4G4C.

5.4.2.1 On-chip Traffic and Energy Efficiency

Other than PE utilization, FlexSA reduces on-chip traffic because it improves in-core input reuse. Figure 5.9 compares the average on-chip input traffic from the GBUF to LBUFs using different core configurations. The red line graphs exhibit normalized traffics to the traffic of 1G1C. The 1G4C and 4G4C configurations increase the average input traffic by 1.66× and 3.13×, respectively, due to the reduced in-core reuse caused by using multiple small systolic arrays. On the other hand, the configurations using FlexSA show similar or even lower GBUF traffic compared to the naive many-small-core designs. 1G1FlexSA lowers GBUF traffic traffic by 41% compared to 1G4C, and by 2% even compared to 1G1C. This 2% traffic saving comes from reusing the stationary inputs between multiple GEMM waves in the VSW and ISW modes, which is not possible with the baseline core design. 4G1FlexSA also saves on average 53% GBUF traffic traffic compared to 4G4C.

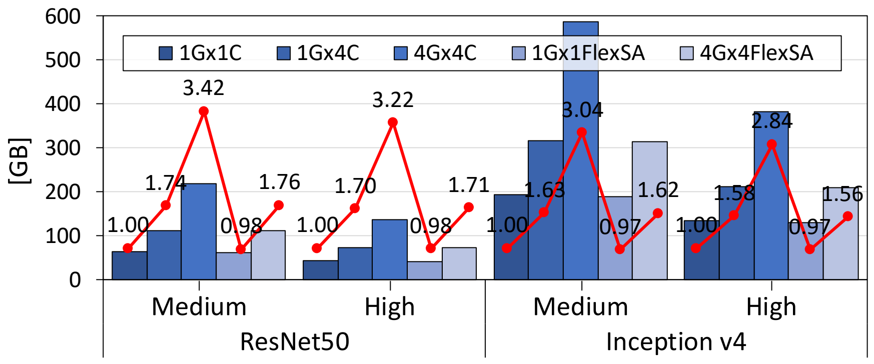
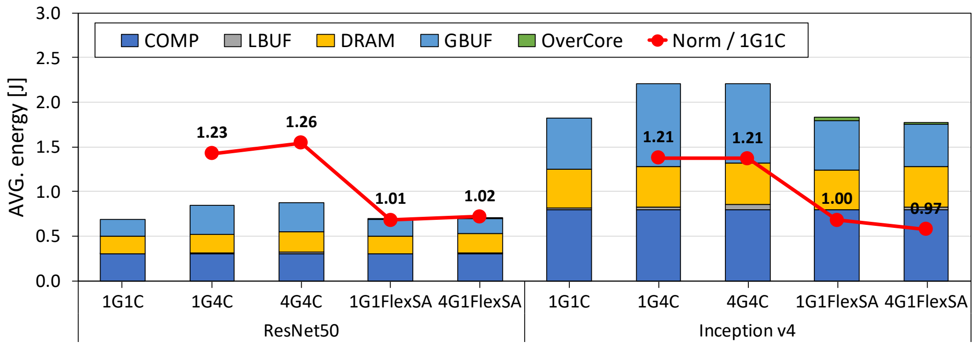
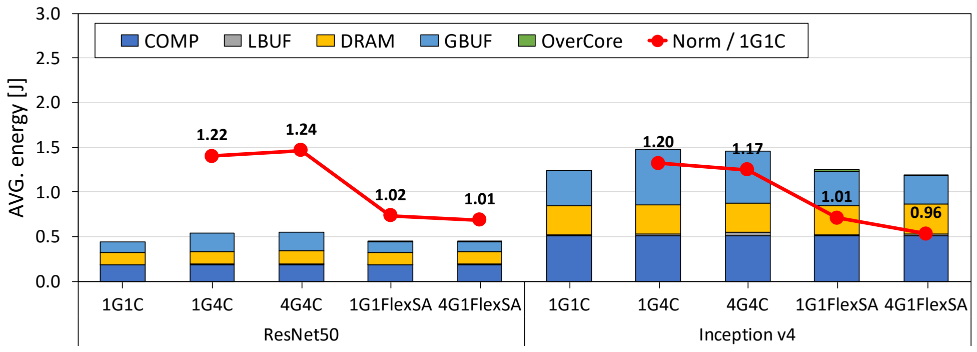


Figure 5.9: On-chip traffic of different core configurations and pruning strengths.

FlexSA reduces on-chip traffic from global buffers and this improves the energy efficiency of training. To show this, I evaluate the average dynamic energy consumed per training iteration using different core configurations (Figure 5.10a). Each bar indicates the breakdown of energy consumed by different accelerator resources: *COMP* indicates the energy consumed by mixed precision MAC (multiply and accumulation) operations. *OverCore* indicates the data transmission energy over a core, which exists only in the configurations using FlexSA. The lines represent the energy increase compared to 1G1C. Because of the increased GBUF traffic, 1G4C and 4G4C exhibit >20% energy increase. The reason 4G4C shows similar energy compared to 1G4C though it requires higher traffic is that average global buffer access energy is lower with the distributed GBUFs. Both FlexSA configurations exhibit similar energy consumption as 1G1C. This high energy efficiency of FlexSA comes from using large GEMM waves in most cases. The additional energy consumed by



(a) Medium pruning strength



(b) High pruning strength

Figure 5.10: Breakdown of dynamic energy consumption for different core configurations and pruning strength.

over-core data transmission is very small for all CNNs and pruning strengths.

5.4.2.2 FlexSA Operating Mode Breakdown

Figure 5.11 shows the breakdown of FlexSA operating modes used by ResNet50 and Inception v4 with different pruning strengths. This shows how frequently large GEMM waves are used to improve in-core reuse and save energy consumption. For ResNet50, the FW accounts for 68% and 86% with 1G1FlexSA and 4G1FlexSA, respectively. VSW and HSW combined are used

25% and 13% of the time with 1G1FlexSA and 4G1FlexSA, respectively. The least efficient mode, ISW, accounts for only 6% and 1% with 1G1FlexSA and 4G1FlexSA. This result indicates that the the inter-core operations of FlexSA are highly used and using only a restricted number of small independent GEMM waves leads to high PE utilization. The results of Inception v4 show similar trends, with the VSW mode selections. This is because many convolution layers in the baseline Inception v4 have a small number of channels and are more efficiently executed with VSW.

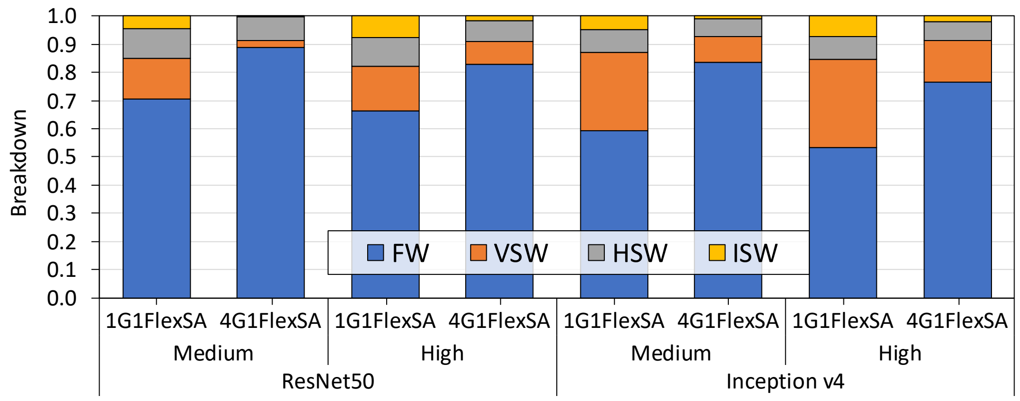


Figure 5.11: FlexSA operating modes breakdown for ResNet50 and Inception v4.

5.5 Summary

In this chapter I tackle the problems in training (partially) pruned CNN models using a high-throughput training accelerator. The GEMMs of channel-pruned CNN models have reduced dimensions, which hurts PE utilization when processed on a training accelerator with large systolic array cores. Although splitting a large core into multiple small cores improves PE utilization, it decreases in-core input reuse, increasing on-chip traffic.

To improve the PE utilization without affecting reuse, I present FlexSA, a flexible systolic array architecture that reconfigures its structure to efficiently process GEMM tiles with different shapes. FlexSA is composed of four small systolic array cores and designing it adds only 1% area compared to naive four-core design. FlexSA supports four different systolic array operating modes. Each FlexSA mode is designed to process GEMM tiles with different shapes, using different inter-core systolic dataflows. For efficient selection of FlexSA operating modes for a given pruned CNN model, I propose FlexSA mode selection heuristics. Overall, FlexSA improves PE utilization of training pruned CNN models by 33% compared to using a single large core and improve on-chip input reuse by 41% compared to small-many-core designs.

Chapter 6

GPU Memory Traffic and Performance Models for Accelerator Design Space Exploration

This chapter introduces a GPU memory traffic model for convolution layers. I first show the need for an accurate data traffic prediction in GPU memory system for effective future GPU design exploration. Then, I explain the memory traffic models for each GPU memory hierarchy (e.g. l1, l2, and DRAM) and a performance model that uses the estimated memory traffic. In the evaluation section, I show how the proposed model helps to fine-tune the GPU resource allocation for efficient CNN performance scaling.

[1]

6.1 Needs of an Accurate Performance Model for CNNs

Estimating the application performance by device design changes provides important guidelines. Given that GPUs are the most commonly used accelerators for DNN training, devising an accurate GPU performance model particularly for DNN applications is important. Based on my observation, to

[1]This chapter was published in International Symposium on Performance Analysis of Systems and Software (ISPASS) in March 2019. Sangkug Lym contributed to this work as the main author.

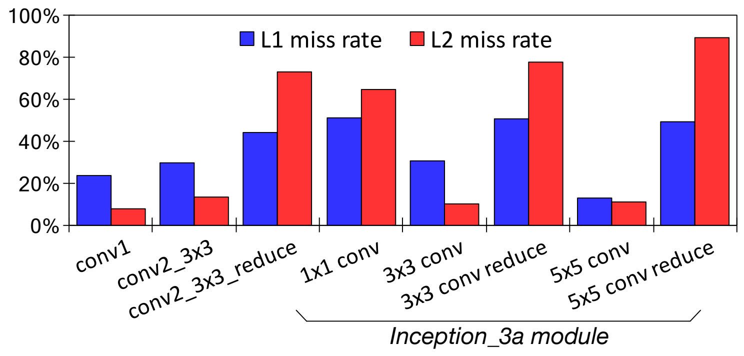


Figure 6.1: L1 and L2 cache miss rates of the convolution layers in Inception v3

predict accurate DNN performance, I should first estimate memory traffic at different levels of GPU memory system hierarchy. Figure 6.1 shows the cache miss rates of convolution layers used in Inception v3 [96] measured on NVIDIA TITAN Xp GPU. Depending on the layer configuration, the L1 miss rate varies from 13% to 50% and L2 miss rate ranges from 8% to 90%. Due to such high traffic variation at different levels of memory hierarchy, the prior performance models fail to accurately predict CNN performance. Particularly, given faster GPU compute throughput scaling than its memory bandwidth, the architecture research for future GPU designs needs accurate data traffic modeling.

6.1.1 Related Work

To analyze the performance bottlenecks of GPU-accelerated data parallel workloads, many GPU performance models have been proposed [41, 60, 61, 114, 100, 90, 105]. In particular, prior models do in-depth analysis of the parallel GEMM and show high accuracy [61, 114]. These prior models predict application performance based on potential execution time bottlenecks such as computation throughput, instruction fetch/issue slots, and global memory bandwidth. However, they do not model the data traffic at each level of GPU memory system hierarchy which depends on each application's memory access and data reuse patterns.

The models proposed by Zhou et al. [114] and Sunpyo et al. [41] include cache miss rate as a parameter but it is naively set to 1. Such assumptions lead to poor performance estimation when the parallel workload has high spatial locality. CuMAPz [55] does consider an application's shared memory utilization by analyzing the device code, but does not consider cache and off-chip memory channel traffic. Wu et al. [105] use machine learning techniques to classify applications to clusters based on GPU resource utilization patterns and they predict performance using the cluster information and the configurations of the target application and GPU. However, without the detailed data traffic in the memory hierarchy, the prediction accuracy is low and it cannot identify precise memory system bottlenecks.

6.2 Background: GEMM Tiling for Efficient Execution on GPU

The convolution GEMM is blocked for efficient execution on a GPU, the most common accelerator for DNN training, as shown in Figure 6.2. The GEMM blocking divides the IFmap (input feature map) matrix with $M \times N$ dimensions into $blk_M \times blk_N$ blocks of CTAs (cooperative thread arrays). The GEMM K dimension is also divided by a blocking factor of blk_K.

Each blocked GEMM execution flow (accumulation in K dimension) consists of three phases: *prologue*, *main loop*, and *epilogue*. During the prologue, each CTA loads blocked IFmaps ($blk_M \times blk_K$) and filter data ($blk_N \times blk_K$) from the global memory (DRAM) to registers. GEMM kernels use input double buffering [59] to overlap these memory loads and the computation routine phases. Thus, the data loaded in the prologue is first fetched to registers and transferred to the shared memory (SMEM) to be used in the first main loop iteration.

The main loops account for the majority of GEMM execution time. The prefetched data in the SMEM in the prior main loop iteration (or prologue) is read and used as the input in the current loop. The computation pipeline multiplies the loaded features and weights then accumulates their results with the results of the prior loop iteration. The data loads, computation, and accumulation operations in the main loop are software pipelined for efficient

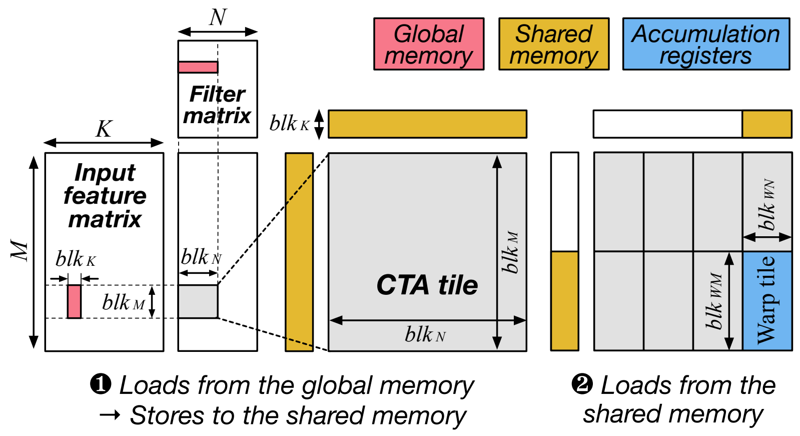


Figure 6.2: Im2col GEMM blocking by cuDNN GPU kernel: ❶ and ❷ show the data movement sequence per GEMM main loop.

resource utilization. After completing the main loops, at the epilogue stage, the accumulated results are written to global memory.

6.3 Memory Traffic Modeling

Designing an analytical model to estimate memory traffic is challenging because: (1) GPU coalesces memory requests to utilize application's spatial locality, (2) different sizes of caches can accommodate different degrees of data locality, (3) caches are shared by multiple threads, CTAs, and cores, and (4) convolution layer workloads with different configurations show different memory access patterns. I model the traffic at each level of GPU memory hierarchy using the granularities of data reuse based on the GEMM kernel blocking factors described in Section **??**. My traffic model incorporates the memory access patterns in the im2col GEMM formed with a specific convolution layer configuration. This is necessary for modeling the complex locality that exists within a single access at each memory level. Also, I reflect the impact of data reuse among CTAs considering the reuse distance of data structures and the CTA scheduling. I use the implicit convolution kernels supported in the cuDNN library, and the NVIDIA Pascal GPU, as the base architecture. I use 32b floating point data precision widely used for neural network model training [33].

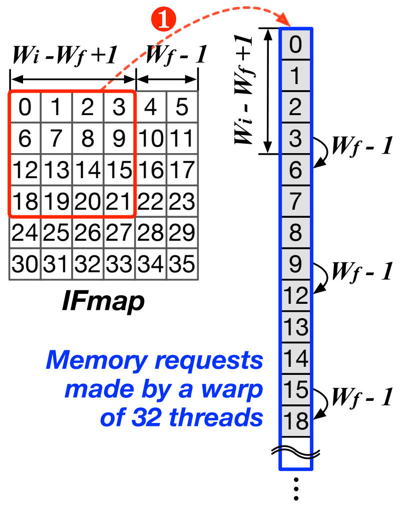


Figure 6.3: IFmap data requested by a single warp per main loop (elements in the blue box).

6.3.1 L1 Cache Traffic

As im2col rearranges the memory access layout, the memory addresses of adjacent IFmap data elements are not continuous. This lowers the spatial locality within each L1 request (individual requests across a warp of 32 threads are coalesced by HW into the minimal number of L1 requests) causing more memory requests than needed. Also, if the memory references are not aligned with the address of the L1 transactions, extra L1 transactions are made. I estimate L1 traffic by calculating the efficiency of L1 requests, which is affected by non-contiguous memory references and address alignment.

Figure 6.3 shows the data layout of a single 6×6 IFmap traversed by one element of a 3×3 filter with stride 1 and its im2col converted form. The number on each data element represents its relative location in the physical memory. As described in Section **??**, IFmap data elements visited by a filter are arranged as a column (➊). One warp requests L1 loads for 32 threads which are a fraction of an IFmap column (blue boxed elements in Figure 6.3). The L1 loads are coalesced within a warp and this coalescing granularity becomes the fundamental access granularity to L1 cache. This corresponds to a L1 cache line size of 128B (4B $\times$ 32) for the Pascal GPU. However, even a warp reads 128B, it cannot be coalesced to a single L1 transaction if data is not continuous: $W_f - 1$ elements are skipped every $W_i - W_f + 1$ elements. If the convolution stride is bigger than 1, data between every two elements are skipped. Also, if the coalesced L1 requests made by a warp are not aligned with the L1 transaction addresses, extra transactions are requested. I use the average L1 load efficiency per warp LE to estimate the average l1 load requests made per warp (Equation 6.1). Finally, I calculate the total L1 traffic by dividing the size of the GEMM input matrices by this l1 load efficiency.

$$\begin{aligned} LE_{IFmap} &= \frac{L1\ requests\ (fully\ aligned)}{Warp} / \frac{L1\ requests\ made}{Warp} \\ &= \frac{L1\ requests\ (fully\ aligned)}{Warp} / \lceil \frac{Elmts\ requested}{Elmts\ used} \times \frac{Bytes}{Warp} \times \frac{L1\ requests}{Bytes} \rceil \end{aligned} \tag{6.1}$$

6.3.2 L2 Cache Traffic

In the im2col GEMM, the IFmap matrix involves many duplicated data accesses, exhibiting high spatial and temporal locality. An L1 cache can capture the reuse within one CTA's IFmap tile but not across active CTAs

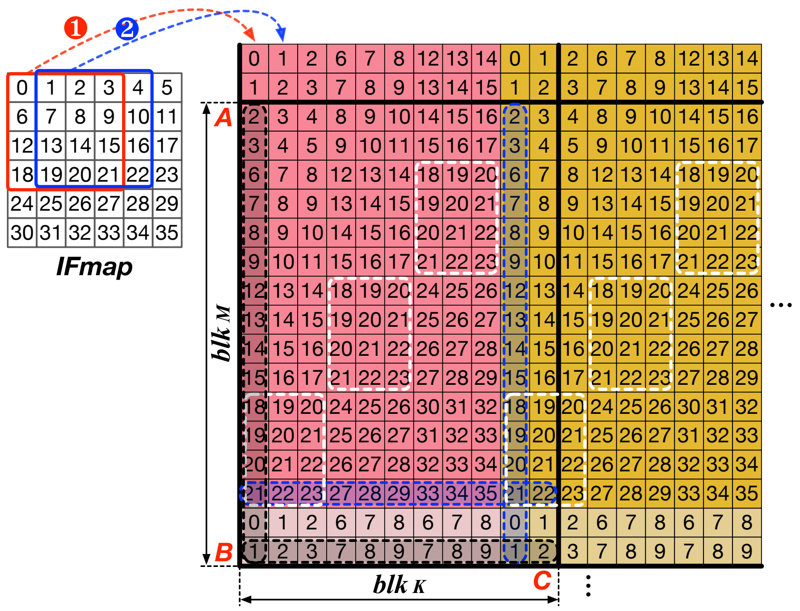


Figure 6.4: IFmap matrix layout of im2col GEMM. The elements in $blk_M \times blk_K$ tile are the input for one main loop.

per SM given its small size (∼32KB). With this assumption, my L2 model estimates the traffic by identifying the unique data requests made in a CTA input tile.

I extend the example used in Section 6.3.1 to explain my L2 traffic model (Figure 6.4). Again, IFmap elements visited by one filter are remapped as a column in the IFmap matrix (❶) and the next column contains the IFmap data traversed by another filter (❷). This access pattern entails large data locality shown by the duplicated data elements in the white doted boxes. As L1 cache captures the locality within each tile, only the unique elements are requested from L2. I use the address range of data within one IFmap tile (difference between the smallest and the largest address) to estimate the size of memory requests to L2 in the tile. Specifically, within each IFmap tile, the accessed address increases from top to bottom and from left to the right.

Given this memory access pattern, the address distances of both the vertical edges of an input block (A and B) and the horizontal edges (B and C) effectively represent the unique data elements requested by the input block. If an input block ranges across multiple data samples or channels, additional load requests are made (elements in the dotted blue boxes), which extends the effective vertical and horizontal distances. I estimate the L2 requests of each input block by calculating the average vertical and horizontal distances.

6.3.3 DRAM Traffic

CTAs in a GEMM kernel share data of both input matrices. As L2 cache is shared by all SMs, the CTAs executed in parallel by all SMs (a CTA batch) can reuse such shared data. However, the specific inter-CTA data reuse depends on CTA scheduling. My DRAM traffic model assumes the GPU uses column-wise CTA scheduling considering im2col GEMM's skinny shape. I estimate the DRAM traffic by identifying the unique data elements within a CTA batch.

CTA tile array of im2col GEMM is tall as its height is set by the product of the height and width of a feature, and the size of the mini-batch. In contrast, its width is relatively narrow and is equal to the number of output channels. Thus, im2col GEMM CTA tile array has a high aspect ratio making it have more CTAs in the column direction than in the row direction. This

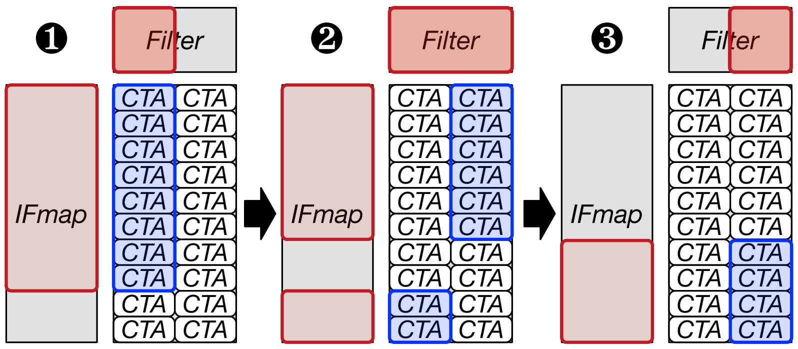


Figure 6.5: IFmap and filter data reference at sequences of processing CTA batches (❶, ❷, and ❸)

increases the chance that CTAs in the same column to be executed in parallel. ❶ to ❸ in Figure 6.5 illustrate the sequences of processing CTA batches. In this example, the GPU has 20 CTAs and there are 8 SMs so the CTA batch size is 8. At each step, the SMs fetch the red boxed IFmap and filter data.

At the sequence of ❶, ❷, and ❸, many CTAs refer to the same filter data, which makes the filter data have a short reuse distance thus increasing the chance of L2 cache locality. Also, each conv layer's total filter size is generally only a few megabytes for recent CNNs [96, 36], so I effectively consider filter data as loaded from DRAM just once. On the other hand, the IFmaps have long rereference distances between columns of CTA tiles in the CTA tile array. Therefore, the overlapping IFmap data across ❶, ❷, and ❸ are fetched twice. This makes the effective IFmap data load counts the same as the columns of CTA tiles in the GEMM. Thus, I effectively estimate the total DRAM traffic by adding the filter data size and the feature data multiplied by the number of columns of a CTA array.

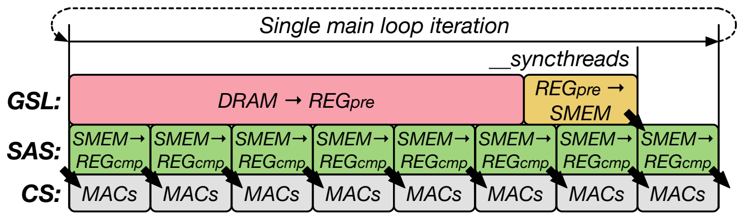


Figure 6.6: Execution time breakdown of a software pipelined GEMM main loop [7]. Three execution streams: global load stream (GLS), shared load stream (SAS), and compute stream (CS)

6.4 GPU Performance Modeling

DeLTA predicts convolution layer's execution time and its performance bottleneck using the memory traffic estimates extracted from my memory traffic model. To identify the execution bottleneck, my model analyzes the compute and memory access streams in the highly software-pipelined GEMM kernel each of which uses different GPU resources. Figure 6.6 shows the execution time breakdown of the software pipelined GEMM main loop [7] (arrows indicate dependencies between execution blocks). Three execution streams proceed in parallel to maximize resource utilization: the global load stream *(GLS)*, shared memory access stream *(SAS)*, and compute stream *(CS)*, that each exercise a different resource. I use this main-loop execution pipelining model to estimate the loop execution time of a convolution layer and to analyze GPU resource utilization.

First, the **global load stream *(GLS)*** loads inputs from the global memory to the registers for prefetch and then to SMEM. GLS execution time is determined by both the latency to load data from the global memory and to store it to SMEM. The total load time consists of the (empty) pipeline latency and the transfer latency. The pipeline latency is the data flight time and it includes cache tagging, buffer pipelining, and all circuit data paths. Pipeline latency is fixed regardless of memory traffic, however, data transfer time increases with the data volume because of the limited bus bandwidth. DeLTA calculates GLS execution time by comparing the load latency from L1, L2, and DRAM using the traffic estimated by the proposed memory traffic model as Equation 6.2. Here, LAT, TpL, and BW indicate load latency, traffic per main loop, and memory bandwidth of each memory level respectively.

$$\begin{aligned} t_{GLS} = &\max\Big(LAT_{L1} + \frac{TpL_{L1}}{BW_{L1}},\ LAT_{L2} + \frac{TpL_{L2}}{BW_{L2}/Num_{SM}}, \\ &LAT_{DRAM} + \frac{TpL_{DRAM}}{BW_{DRAM}/Num_{SM}}\Big) \end{aligned} \tag{6.2}$$

Second, the **shared memory access stream *(SAS)*** loads the prefetched data from the SMEM to registers. The execution time of SAS is bound by the SMEM bandwidth and the data volume of both the SMEM loads and the SMEM stores (from GLS). This is because the loads from SMEM share the same data path with the stores to SMEM. The data volume of the SMEM stores is set by the CTA blocking factors $(blk_N + blk_M) \times blk_K$. Then, the SMEM loads transfer the stored data to registers for computation of each warp Figure 6.2, thus their data volume is set by the warp blocking factors $(blk_{WN} + blk_{WM}) \times blk_K$ multiplied by the number of warps per CTA (Num_{warps}). I calculate the SAS execution time per main loop by dividing the data volume of the SMEM stores and SMEM loads by their respective bandwidth Equation 6.3.

$$t_{SAS} = \frac{(blk_M + blk_N) \times blk_K}{BW_{SMEM\_ST}} + \frac{(blk_{WM} + blk_{WN}) \times blk_K \times Num_{warps}}{BW_{SMEM\_LD}} \tag{6.3}$$

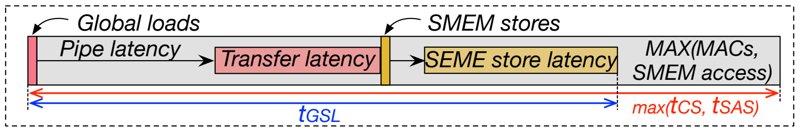
Finally, the **compute stream** ***(CS)*** performs matrix multiplication and accumulation (MAC) operations and its execution time is determined by the compute throughput of each SM. The GEMM kernel interleaves pieces of SAS over CS to hide the SMEM access latency. Therefore, if CS is the execution time bottleneck, the loop execution time is the number operations per main-loop divided by MAC bandwidth.

6.4.1 Multi-CTA Interleaving

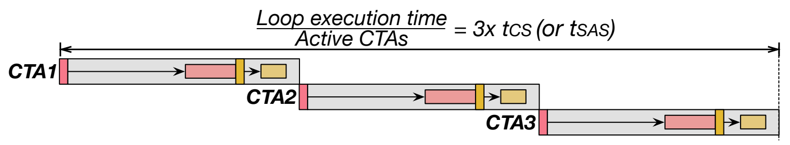
When an SM has multiple active CTAs, t_{GLS} can be further hidden by t_{SAS} or t_{CS} phases of other CTSs. Especially, the execution time prediction in the context of CTA interleaving is important for GPU with high compute throughput. This is because t_{CS} per main loop can be shorter than t_{GLS} thus needing multiple CTAs to hide the load latency.

Given multiple CTAs to interleave, I model the execution time of the active CTAs with four potential resource bottleneck cases (Figure 6.7). The examples (case 1–4) have multiple CTAs to interleave and each row is the time slot for one CTA. I use the simple CTA loop execution time model depicted in Figure 6.7a as the base. The computes and SMEM accesses are spread over a loop iteration (gray background color).

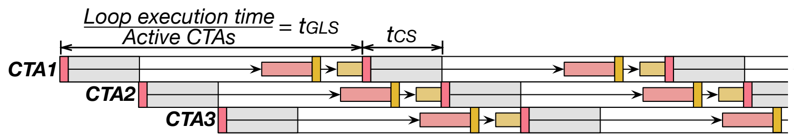
In case 1, the loop execution time per CTA is bottlenecked by t_{CS} or t_{SAS} so I calculate the loop execution time of a CTA batch by adding $\max(t_{CS}, t_{SAS})$ of all active CTAs per SM. Second, if t_{CS} of all CTAs is shorter than t_{GLS} but longer than the memory transfer latency, the loop execution time per CTA batch equals to a single CTA's t_{GLS} (case 2). In this case, each SM has insufficient number of CTAs to hide the loads from the global memory so SM resources are wasted until the entire data arrives for the next loop iteration. Third, if an SM has many CTAs to interleave, t_{GLS} can be completely



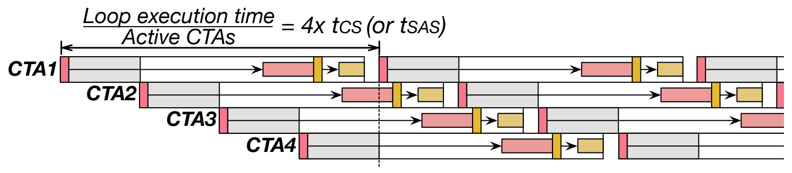
(a) Simplified CTA GEMM main loop execution timing model



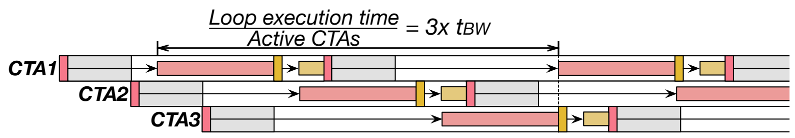
(b) Case 1. $\max(t_{CS}, t_{SAS}) \geq t_{GLS}$



(c) Case 2. $t_{GLS} \geq t_{CS} \times Num_{ACT\_CTA}$



(d) Case 3. $\max(t_{CS}, t_{SAS}) \times Num_{ACT\_CTA} \geq t_{GLS}$



(e) Case 4. $\max(t_{L1\ BW}, t_{L2\ BW}, t_{DRAM\ BW}) \geq t_{CS}$

Figure 6.7: GEMM loop execution model of active CTAs with different GPU resource bottlenecks.

hidden by the time for computation or SMEM access (case 3). Finally, if the memory bandwidth becomes the bottleneck due to high data transfer time (red portion), CTA batch loop execution time is mostly dependent on the data transfer time of the active CTAs (case 4). DeLTA estimates the loop time by comparing the four possible performance bottlenecks.

Eventually, I derive the total execution time of all CTAs allocated per SM by different execution constraints (Equation 6.4, Equation 6.5, and Equation 6.6). Equation 6.4 is used to compute the execution time of case 1 and 3. Equation 6.5 and Equation 6.6 are used for case 2 and 4 respectively (Equation 6.6 derives three sub-results each for L1, L2, and DRAM BW). Then, the largest result of all cases becomes the per-SM execution time and its performance bottleneck.

$$t_{MAC_{(sm)}(SMEM_{(sm)})} = \big(t_{CS}(t_{SMEM}) \times \frac{K}{blk_K} + t_{Epilogue}\big) \times \frac{Num_{CTA}}{Num_{SM}} \tag{6.4}$$

$$t_{DRAM\_LAT_{(sm)}} = \Big(t_{GLS} + \max\Big(\frac{t_{CS}}{blk_K}, \frac{t_{SAS}}{blk_K}\Big)\Big) \times \frac{K}{blk_K} + t_big) \times \frac{N_{CTA}/N_{SM}}{N_{ACT\_CTA}} \tag{6.5}$$

$$t_{MEM\_BW_{(sm)}} = \big(\max(t_{L1\_BW}, t_{L2\_BW}, t_{DRAM\_BW}) \times \frac{K}{blk_K} + t_{Epilogue\_bottleneck}\big) \times \frac{Num_{CTA}}{Num_{SM}} \tag{6.6}$$

6.5 Evaluation

6.5.1 Evaluation Environment

Device Specification. I compare the data traffic and performance estimates to the measured data on two Pascal GPUs (TITAN Xp and P100) [74] and one Volta GPU (V100) [76] (Table 6.1). Since the memory access latencies and bandwidths are not specified by NVIDIA, I measure the access latency and bandwidth to L1, L2, and DRAM using microbenchmarks (both new and from prior work [70]).

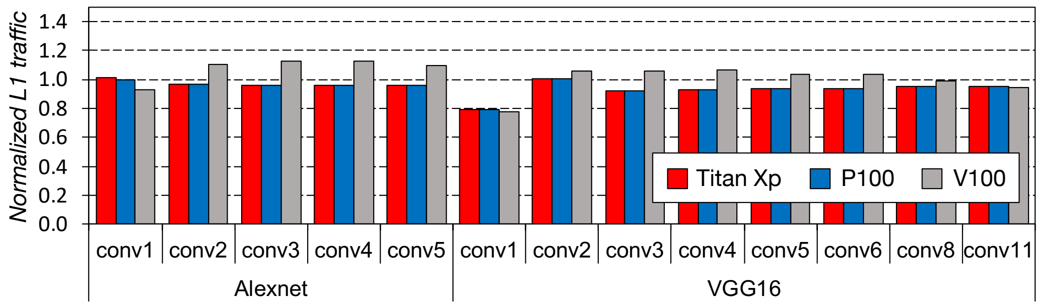
Table 6.1: GPU device specifications

Specifications	Pascal TITAN Xp	Pascal P100	Volta V100
Num_{SM}	30	56	84
Core clock	1.58 GHz	1.2GHz	1.38GHz
BW_{MAC} $(FP32)$	12134 GFLOPS	8602 GFLOPS	14837 GFLOPS
$Size_{REG}$	256 KB/SM	256 KB/SM	256 KB/SM
$Size_{SMEM}$	96 KB/SM	64 KB/SM	$\leq$94 KB/SM
BW_{L1}	92 GB/s/SM	38.1 GB/s/SM	94.1 GB/s/SM
BW_{L2}	1051 GB/s	1382 GB/s	2167 GB/s
BW_{DRAM}	450 GB/s	550 GB/s	850 GB/s
$Size_{L2}$	3MB	4MB	6MB

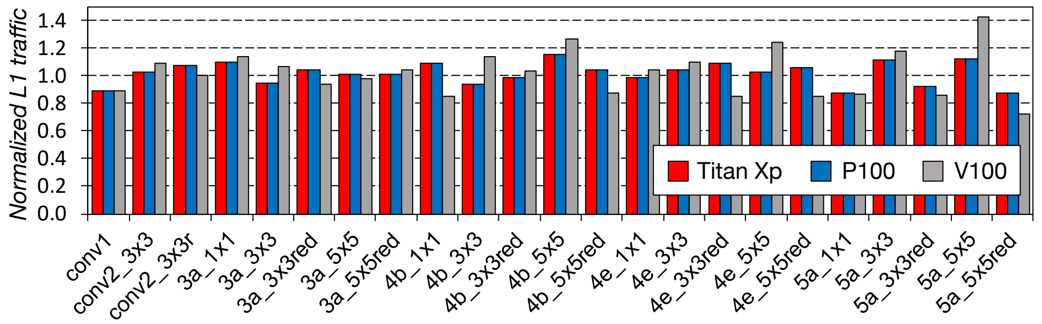
Benchmarks. I evaluate DeLTA on the convolution layers of four popular CNNs (AlexNet [57], VGGNet [92], GoogLeNet (Inception v3) [96], and ResNet [36]) used for ImageNet dataset training and prediction [88]. Because many convolution layers in these CNNs share configurations, I show the results on the unique subset. Unless specified, a mini-batch size of 256 is used for all evaluated layers. I use cuDNN ConvolutionForward API with IMPLICIT PRECOMP GEMM algorithm to run convolution layers compiled with CUDA v8.0 and cuDNN v7.0.

6.5.2 Memory Traffic Model

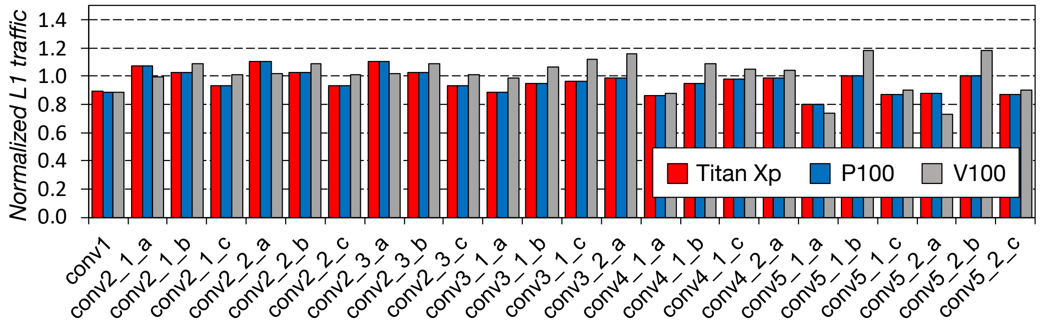
Figure 6.8, Figure 6.9, and Figure 6.10 show DeLTA estimates of data traffic for L1, L2, and DRAM normalized to the measurement of three different GPUs for all unique layer configurations of the 4 evaluated CNNs. Both



(a) AlexNet and VGG16

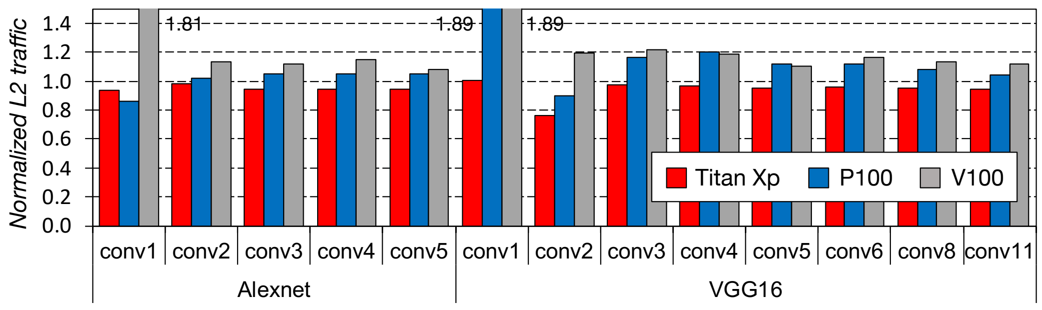


(b) GoogLeNet

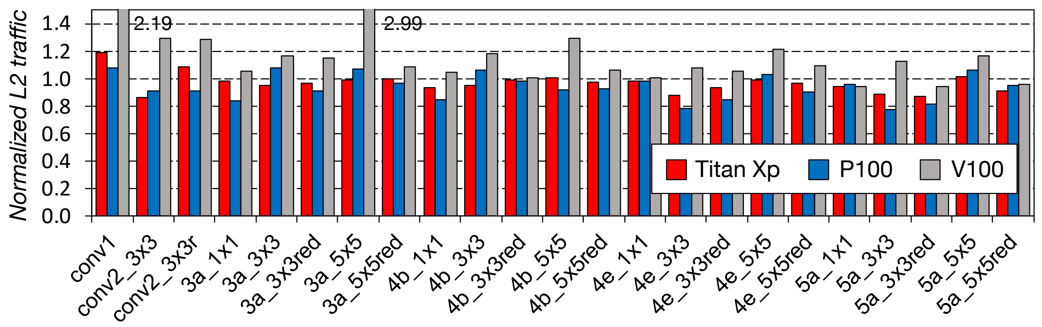


(c) ResNet

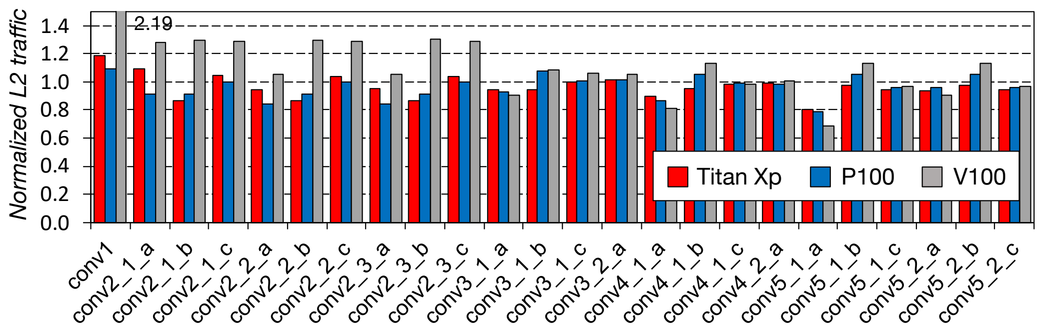
Figure 6.8: L1 traffic estimates of the unique convolution layers by DeLTA normalized to the measured values on three different GPUs



(a) AlexNet and VGG16

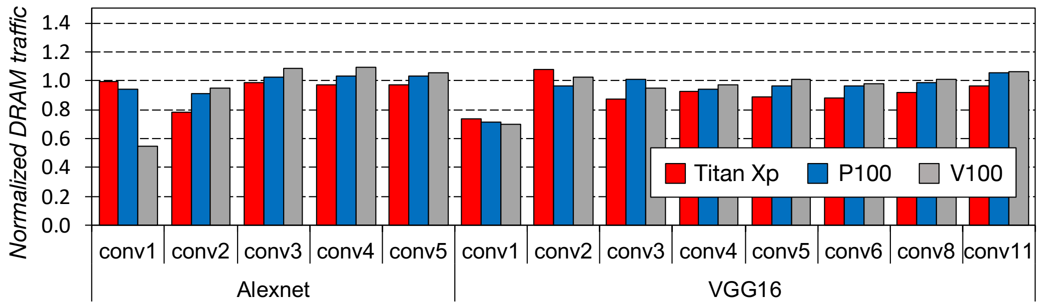


(b) GoogLeNet

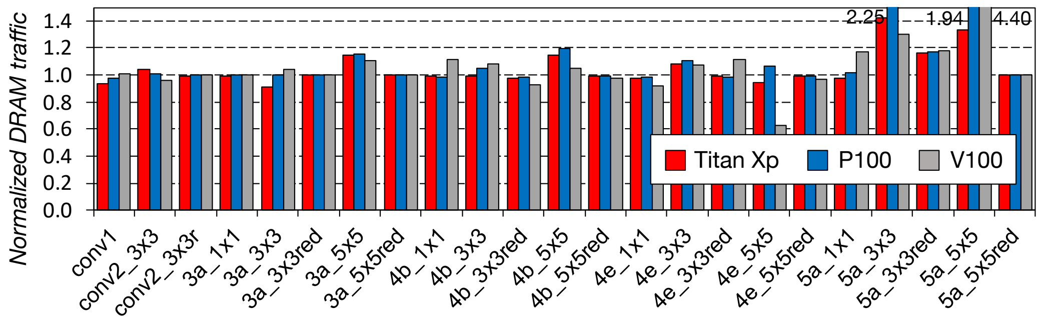


(c) ResNet

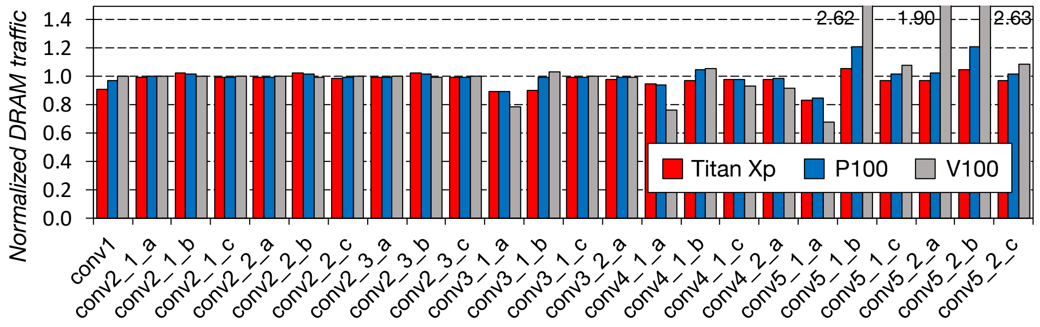
Figure 6.9: L2 traffic estimates of the unique convolution layers by DeLTA normalized to the measured values on three different GPUs



(a) AlexNet and VGG16



(b) GoogLeNet



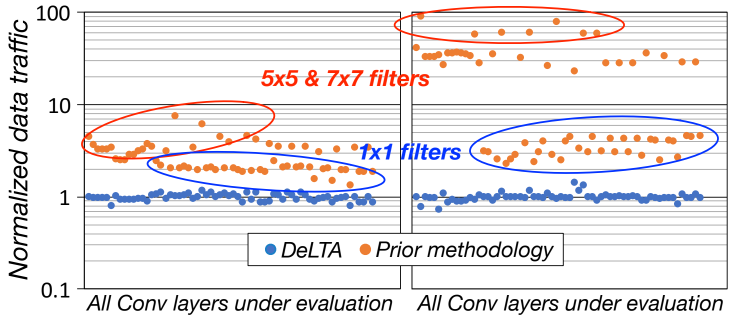
(c) ResNet

Figure 6.10: DRAM traffic estimates of the unique convolution layers by DeLTA normalized to the measured values on three different GPUs

TITAN Xp and P100 use the same kernels and have an L1 request size of 128B so the measured and predicted L1 traffic is the same for both. DeLTA shows high accuracy with a GMAE (geometric mean absolute error) of 4.6% (7.9% standard deviation). I am unsure of the L1 request size for Volta and experimented with 32B, 64B, and 128B granularity settings for DeLTA. I observed the best match to measurements with 32B L1 requests and DeLTA matches measured GV100 results with a GMAE (Geometric Mean Absolute Error) of 6.9% (13.3% stdev).

The modeled L2 traffic has a larger variation in error than L1 traffic. L2 traffic is the result of misses in L1 and my model makes the simplifying assumption that there is no overlap in data access to L1 between different CTAs. I hypothesize that the greater error is indeed the result of multiple concurrent CTAs reusing some data in L1. Two observations from the result strongly support my hypothesis. First, DeLTA over-estimates traffic primarily for layers that have larger features, which lead to greater opportunity for reuse between CTAs. Second, I see larger modeling errors for V100 than P100 and Titan XP. The V100 architecture has larger L1 caches (unified with SMEM) [76] that can also contribute to more reuse across CTAs. Overall, DeLTA is still quite accurate with the GMAE and standard deviation (in parentheses) being 4.2% (7.2%) for Titan XP, 6.2% (14.4%) for P100, and 12.4% (25.0%) for V100.

The modeled DRAM traffic is very accurate overall with a few notable outliers. Some of the layers in GoogLeNet and and ResNet have very



(a) L2 traffic comparison (b) DRAM traffic comparison

Figure 6.11: L2 and DRAM traffic estimates by DeLTA and prior methodology normalized to TITAN Xp measurement

small memory footprints that can completely fit within the L2 cache. The profiler I use to measure results reports anomalous numbers for these layers that suggest the impossible scenario where less data is read from DRAM than the actual footprint, leaving DeLTA with a large over-estimation. The other source of large modeling error relates to L2 cache behavior and CTA scheduling. DeLTA underestimates DRAM traffic for VGG16-conv1, GoogLeNet-4e_5×5, and a few ResNet layers. My analysis indicates that these errors result from DeLTA identifying potential data reuse in L2 between CTAs that is not exploited by the hardware. Overall, the DRAM traffic model shows small GMAE (with standard deviation) of 2.8% (10.3%) for Titan Xp, 6.2% (14.4%) for P100, and 10.2% (9.2%) for V100 (without the anomalous measurements).

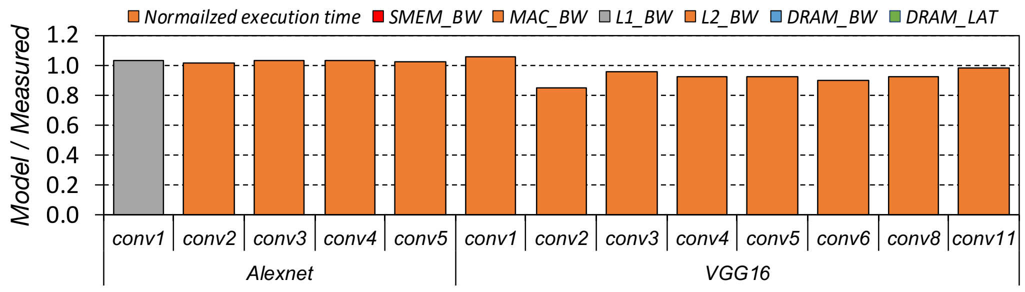
Memory Traffic Comparison with Prior Models. Figure 6.11 compares the normalized data traffic for all unique convolution layers under evaluation for DeLTA and the prior models [114, 41]. As the prior models assume 100% cache miss rates for both levels of caches, I apply the L1 load traffic to both

L2 and DRAM. Both L2 and DRAM traffic assumed by the prior models are far from the measurements because these prior models ignore the high data reuse in the convolution layers. The deviation is relatively small for layers with 1×1 filters due to the low data reuse but the large filters have very large errors. These errors are multiple factors (up to nearly 100×) larger than those of DeLTA, and lead to wrong conclusions about performance bottlenecks of modern GPUs with their large arithmetic to memory throughput ratios.

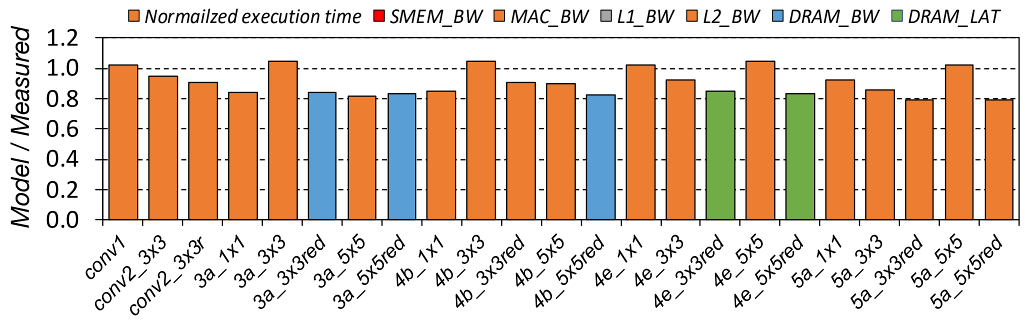
6.5.3 Performance Model

Figure 6.12 shows the performance estimations vs. the measured data on TITAN Xp, along with their bottlenecks where the GMAE is 6.0%. Although highly accurate, DeLTA underestimates the execution time for some layers regardless of their bottlenecks. One major reason is that the estimated memory traffic is uniform across each CTA's GEMM main loop but the actual data traffic is not. For example, if the data fetched by one iteration is reused in the next, the first iteration execution time is bound by the data loads but the second iteration is constrained by arithmetic throughput. Such non-uniform execution time bottlenecks make DeLTA provide conservative estimates that represent worst-case performance.

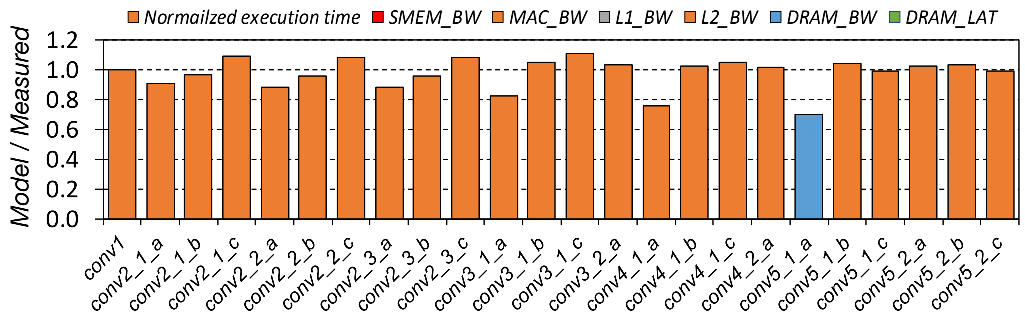
My evaluation shows that arithmetic throughput is the major performance bottleneck (90% of evaluated layers), which is expected due to the high data reuse of im2col GEMM. I also observe that some layers are bottlenecked by other resources. L1 BW restricts the first convolution layer of AlexNet due



(a) AlexNet and VGG16



(b) GoogLeNet



(c) ResNet

Figure 6.12: Conv layer execution time estimates by DeLTA normalized to TITAN Xp's and their performance bottlenecks

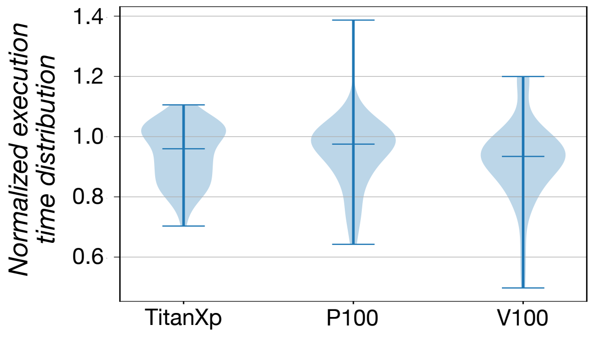


Figure 6.13: Estimated GPU performance: each normalized to measurement using different GPUs

to its poor L1 transaction efficiency. Many layers in GoogLeNet are bottlenecked by DRAM BW or latency. The layers bottlenecked by DRAM latency do not have enough CTAs to hide the load latency. The layers restricted by DRAM BW have more CTAs to interleave thus saturate the DRAM channels.

Performance Estimation for Different GPUs. Figure 6.14.a compares the performance estimation distribution for the three GPUs. DeLTA estimates performance best for Titan XP, but the overall accuracy is quite good For

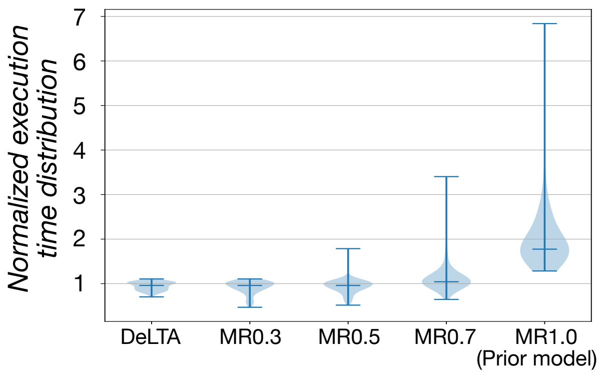


Figure 6.14: Comparison to the fixed miss rate (MR) models.

P100 and V100 as well with a robust low-variance estimation (10% standard deviation). The outliers correspond to those layers for which data traffic is poorly estimated because of dynamic behavior, as explained above.

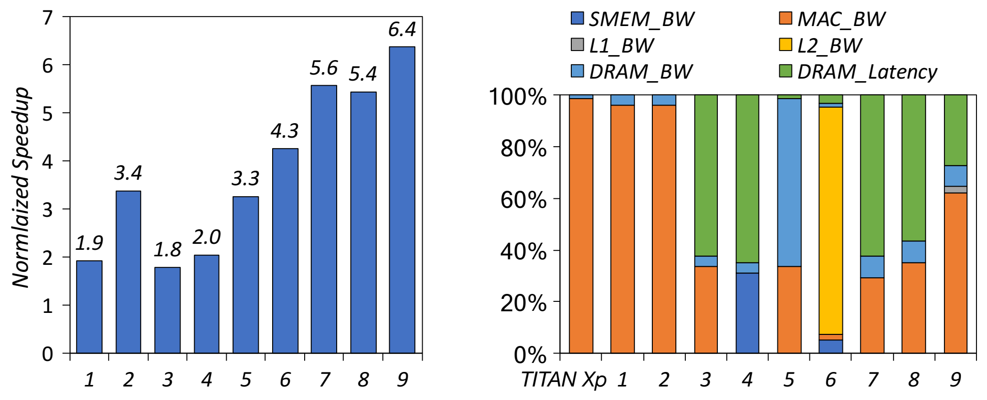
Comparison to Fixed Miss Rate Models. Figure 6.14.b compares the normalized performance estimation results of DeLTA and the models; while prior work advocated using a 1.0 miss rate, I sweep a range of miss rates in the figure. Compared to DeLTA, the other models show wider estimation errors and a larger number of outliers. With the 1.0 miss rate advocated by the prior models, the layer execution time is over-predicted by 1.8× on average and up to 7×. The prediction error for the models using fixed miss rates becomes significantly larger when compute throughput scales as many layers become memory system resource bottleneck.

6.5.4 Fine-grain GPU Performance Scaling Study

I use DeLTA to explore GPU designs for efficient CNN performance scaling with less HW resources. I use the entire 152 convolution layers in ResNet152 to evaluate the potential speedup over the Titan Xp baseline. As almost all compute kernels of ResNet are convolution layers, its performance is dominated by their execution time. Table 6.2 shows the design options used in my experiment. Option 1 and 2 represent the conventional way to improve GPU performance, which keeps SM resources constant and scales the number of SMs and L2 and DRAM BW. These options are expensive as each extra SM involves the entire SM resources such as registers, SMEM capacity and BW, and L1. For the other design options, I use the resource bottleneck information from DeLTA to minimally scale independent resources for efficient performance gain.

Table 6.2: GPU design options. Each column indicates a GPU design choice and xX indicates the magnitude of increase.

Resources	TitanXp	1	2	3	4	5	6	7	8	9
N_{SM}	1X	2X	4X	1X	1X	1X	1X	1X	2X	1X
MAC_{BW} / SM	1X	1X	1X	2X	4X	4X	6X	8X	4X	8X
$REGS_{SIZE}$ / SM	1X	1X	1X	1X	1X	2X	2X	3X	2X	3X
$SMEM_{SIZE}$ / SM	1X	1X	1X	1X	1X	2X	2X	3X	2X	3X
$SMEM_{BW}$ / SM	1X	1X	1X	1X	1X	2X	2X	3X	2X	3X
$L1_{BW}$ / SM	1X	1X	1X	1X	1X	1.5X	2X	2X	2X	2X
$L2_{BW}$	1X	1.5X	2X	1X	1X	1.5X	1.5X	2X	2X	2X
$DRAM_{BW}$	1X	1.5X	2X	1X	1X	1.5X	2X	2X	2X	3X
CTA tile H,W	128	128	128	128	128	128	128	256	256	256



(a) Speedup normalized to TITAN Xp

(b) Performance bottleneck distribution of each design option

Figure 6.15: GPU resource scaling and speedup of convolution layers in ResNet

Figure 6.15.a shows the normalized performance gain for the different design options with their performance bottlenecks shown in Figure 6.15.b. DeLTA predicts that increasing the number of SMs by 2× and 4× along with

L2 and DRAM BW improves ResNet forward propagation performance by 1.9× and 3.4×, respectively. The limiting factor is memory BW increase, which restricts pipelined GEMM epilogue and some layers with larger memory BW requirements. Given convolution layers are compute throughput hungry, options 3 and 4 increase only the per core arithmetic throughput by adding more MAC units. However, their performance headroom is only 2× with most layers bottlenecked by DRAM BW and SM resources.

Based on these observations, option 5 minimally increases resources to avoid bottlenecks, showing similar performance gains to option 2 with far lower hardware resource increases. Option 6 further increases compute throughput but DeLTA shows that now L2 BW becomes the limiter. From option 7 to 9, I increase the size of the GEMM tiles given the high compute throughput to memory bandwidth ratio. Such GEMM parameter choices are only beneficial for GPU designs with high arithmetic throughput, otherwise the performance is restricted by the memory system BW. Design option 8 increases the number of SMs by 2×, but DeLTA shows that increasing DRAM BW is more beneficial than doubling the SM resources (option 9). As such, using DeLTA and model of the potential hardware resource costs, GPU architecture can be more efficiently optimized for CNN performance.

6.6 Discussion: Generalization of DeLTA

Back-propagation Convolution Layer Kernel Modeling. In this dissertation, I evaluate the convolution kernels only in network forward propagation

(FPROP). However, in back-propagation, each convolution layer involves two GEMM kernels that compute the gradients of loss w.r.t the weights (WGRAD) and the upstream loss gradients (DGRAD) respectively. DGRAD convolution kernels involve similar memory access patterns of the FPROP convolution kernels except for the input matrix transpose. I simply applied DeLTA to DGRAD convolution kernels and confirmed high traffic and performance prediction accuracy using NVIDIA TITAN Xp GPU. However, WGRAD convolution kernels have different GEMM dimensions because the GEMM input and accumulation dimensions are switched. Thus, WGRAD convolution kernels of early layers have a few CTAs due to their small GEMM output size of $[input\_channels \times filter\_size] \times [output\_channels]$ but have deep accumulation depth of $batch \times OFmap\_size$. This causes load imbalance across SMs on GPU and inefficient DRAM bandwidth utilization and eventually DeLTA fails to estimate accurate memory traffic and performance. The WGRAD kernel can be optimized for better resource utilization by splitting the GEMM accumulation dimension into multiple CTAs followed by inter-CTA reduction and DeLTA can be easily tunned for such software changes.

Application to General Data-parallel Kernels. DeLTA is limited to only the data-parallel kernels without indirect memory accesses. Given the memory access patterns and the blocking factors of a data-parallel application are known by analyzing the kernel codes, the memory traffic modeling mechanisms of DeLTA can easily be used to estimate its memory traffic. However, if the working set of applications is smaller than or similar to the size of GPU caches, the prediction accuracy will significantly drop because DeLTA mainly relies on the spatial locality of each blocked memory load. I also model this caching impact to the memory traffic using polynomial regression using the measured inputs of three different GPUs. However, they are limited to a particular application so the trained parameters for convolution kernels should be retrained for different applications and they are not reliably reusable for different GPU generations.

6.7 Summary

In this chapter, I introduce DeLTA, a GPU memory traffic model that accurately estimates the memory traffic of convolution layers at all levels of the GPU system. I analyze the complex memory access patterns of im2col GEMM, the most-commonly used convolution algorithms for accelerating CNNs on GPUs. I also study the GEMM execution blocking mechanism that affects the spatial data reuse at different memory levels. I use the estimated traffic to model the expected performance of a convolution layer executed on different GPUs, where all important GPU characteristics are parameterized to enable the rapid identification of bottlenecks and the evaluation of design tradeoffs. I eventually show how my models can be used to explore the design space and better tune GPU resource provisioning for CNNs when compared to equally scaling all resources or ignoring the need for higher memory bandwidth as arithmetic throughput increases.

Chapter 7

Conclusion

High computation throughput and memory bandwidth demands for fast CNN processing have opened a new heyday of data-parallel processors and domain specific accelerators. This dissertation presents memory bandwidth- and computation-efficient CNN model training methods by co-optimizing training workload scheduling, learning algorithms, and the accelerator architecture. I demonstrate that the proposed memory bandwidth-efficient training speeds up training even with cheaper hardware resources and the proposed computation-efficient training reduces the computations needed for training with minor accuracy impact.

Serializing Mini-batched CNN Training. I explore the inter-layer data reuse opportunities in CNN training and show that the conventional mini-batch SGD training used with an existing accelerator cannot utilize this locality. Then, I propose a partial mini-batch serialization that not only removes most redundant memory accesses by capturing the inter-layer locality but supports high PE utilization by maintaining high data parallelism across layers. To apply the partial mini-batch serialization to modern CNN models, I adapt Group Normalization and avoid layer-wise synchronization. Finally, I introduce a training accelerator that adopts the architectural optimizations to support gap-less matrix operation pipelining given the proposed training scheduling changes.

Structured Model Pruning During Training. I present a method to prune model parameters during training to gradually reduce computation cost of training. This is based on my observation that pruning non-critical parameters during training has negligible impact to the learning quality. I structurally prune these parameters using group lasso regularization to provide high data parallelism without involving complex data indexing, which maintains high compute unit utilization. For the practical implementation of this training method to modern CNN models, I propose key algorithmic optimizations. The proposed optimizations remove the need for the memory accesses caused by tensor reshaping, dynamically increase the training mini-batch size to increase both the data parallelism and reduce both memory accesses and inter-accelerator communication.

Flexible Systolic Array Architecture for Efficient Pruned CNN Model Training. I introduce the problem that training a pruned CNN models on a large systolic core causes severe PE underutilization. To mitigate this PE underutilization, I propose a flexible systolic array architecture that it reconfigures its structure to efficiently process GEMM tiles with different shapes. To provide this reconfigurability, I use four small cores to support three intercore operating modes, each designed for mapping GEMM tiles with different shapes efficiently to systolic arrays. I also propose heuristics to map the tiled GEMM to inter-core operating modes that leads to near-optimal PE utilization and high input reuse. The proposed systolic array architecture and its efficient utilization achieves as high PE utilization as using multiple independent cores and as high input reuse as using a single large core.

Data Traffic Modeling for Efficient CNN Performance Scaling. I develop a GPU performance model to estimate accurate CNN execution time on future GPU designs. Given that efficient CNN acceleration requires a balance between computing throughput and memory bandwidth, I propose a memory traffic model that accurately estimates the data traffic at a different level of GPU memory hierarchy. The proposed memory traffic model involves an in-depth analysis of the memory access patterns of data-parallel convolution layer kernels and the spatial locality coupled with the memory access granularity of different memory levels. I show that the proposed GPU performance model can be used to fine-tune the hardware resources for cost-efficient CNN performance scaling.

Bibliography

[1] Tor M Aamodt, Wilson WL Fung, I Singh, A El-Shafiey, J Kwa, T Hetherington, A Gubran, A Boktor, T Rogers, A Bakhoda, et al. Gpgpu-sim 3. x manual, 2012.

[2] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. Extremely large minibatch sgd: training resnet-50 on imagenet in 15 minutes. *arXiv preprint arXiv:1711.04325*, 2017.

[3] Jose M Alvarez and Mathieu Salzmann. Learning the number of neurons in deep networks. In *Advances in Neural Information Processing Systems*, pages 2270–2278, 2016.

[4] Jose M Alvarez and Mathieu Salzmann. Compression-aware training of deep networks. In *Advances in Neural Information Processing Systems*, pages 856–867, 2017.

[5] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-layer cnn accelerators. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–12. IEEE, 2016.

[6] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. Deep speech 2: End-to-end

speech recognition in english and mandarin. In *International conference on machine learning*, pages 173–182, 2016.

[7] Julien Demouth Andrew Kerr, Duane Merrill and John Tran. Cutlass: Fast linear algebra in cuda c++. *NVIDIA Developer Blog*, 2017.

[8] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Structured pruning of deep convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 13(3):32, 2017.

[9] Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Prithviraj Sen, Alexandre V Evfimievski, and Niketan Pansare. On optimizing operator fusion plans for large-scale machine learning in systemml. *Proceedings of the VLDB Endowment*, 11(12):1755–1768, 2018.

[10] Léon Bottou, Frank E Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *Siam Review*, 60(2):223–311, 2018.

[11] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An analysis of deep neural network models for practical applications. *arXiv preprint arXiv:1605.07678*, 2016.

[12] Ke Chen, Sheng Li, Naveen Muralimanohar, Jung Ho Ahn, Jay B Brockman, and Norman P Jouppi. Cacti-3dd: Architecture-level modeling for 3d die-stacked dram main memory. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 33–38. EDA Consortium, 2012.

[13] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[14] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM Sigplan Notices*, 49(4):269–284, 2014.

[15] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.

[16] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2019.

[17] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE Computer Society, 2014.

[18] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model

compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282*, 2017.

[19] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[20] Dipankar Das, Sasikanth Avancha, Dheevatsa Mudigere, Karthikeyan Vaidynathan, Srinivas Sridharan, Dhiraj Kalamkar, Bharat Kaul, and Pradeep Dubey. Distributed deep learning using synchronous stochastic gradient descent. *arXiv preprint arXiv:1602.06709*, 2016.

[21] Jeff Dean. Machine learning for systems and systems for machine learning. 2017.

[22] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[23] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[24] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. Shidiannao: Shifting vision processing closer to the sensor. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 92–104. ACM, 2015.

[25] Jiashi Feng and Trevor Darrell. Learning the structure of deep convolutional networks. In *Proceedings of the IEEE international conference on computer vision*, pages 2749–2757, 2015.

[26] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. A configurable cloud-scale dnn processor for real-time ai. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pages 1–14. IEEE Press, 2018.

[27] Hiroshi Fuketa, Koji Hirairi, Tadashi Yasufuku, Makoto Takamiya, Masahiro Nomura, Hirofumi Shinohara, and Takayasu Sakurai. Minimizing energy of integer unit by higher voltage flip-flop: Vddmin-aware dual supply voltage technique. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(6):1175–1179, 2013.

[28] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. Tetris: Scalable and efficient neural network acceleration with 3d memory. *ACM SIGOPS Operating Systems Review*, 51(2):751–764, 2017.

[29] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[30] Edouard Grave, Armand Joulin, and Nicolas Usunier. Improving neural language models with a continuous cache. *arXiv preprint arXiv:1612.04426*, 2016.

[31] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al. Ese: Efficient speech recognition engine with sparse lstm on fpga. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 75–84. ACM, 2017.

[32] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 243–254. IEEE, 2016.

[33] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[34] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

[35] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017.

[36] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[37] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European conference on computer vision*, pages 630–645. Springer, 2016.

[38] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 784–800, 2018.

[39] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *International Conference on Computer Vision (ICCV)*, volume 2, 2017.

[40] Brian Hickmann, Andrew Krioukov, Michael Schulte, and Mark Erle. A parallel ieee p754 decimal floating-point multiplier. In *Computer Design, 2007. ICCD 2007. 25th International Conference on*, pages 296–303. IEEE, 2007.

[41] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 152–163. ACM, 2009.

[42] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[43] Hengyuan Hu, Rui Peng, Yu-Wing Tai, and Chi-Keung Tang. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. *arXiv preprint arXiv:1607.03250*, 2016.

[44] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. *arXiv preprint arXiv:1709.01507*, 7, 2017.

[45] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *CVPR*, volume 1, page 3, 2017.

[46] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[47] Stanislaw Jastrzkebski, Zachary Kenton, Devansh Arpit, Nicolas Ballas, Asja Fischer, Yoshua Bengio, and Amos Storkey. Three factors influencing minima in sgd. *arXiv preprint arXiv:1711.04623*, 2017.

[48] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, et al. Highly scalable deep learning training system with mixed-precision:

Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205*, 2018.

[49] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[50] Joint Electron Device Engineering Council. *High Bandwidth Memory (HBM) DRAM, JESD235A*, Jan. 2016.

[51] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12. ACM, 2017.

[52] Wonkyung Jung, Daejin Jung, Sunjung Lee, Wonjong Rhee, Jung Ho Ahn, et al. Restructuring batch normalization to accelerate cnn training. *arXiv preprint arXiv:1807.01702*, 2018.

[53] Andrew B Kahng, Bin Li, Li-Shiuan Peh, and Kambiz Samadi. Orion 2.0: A fast and accurate noc power and area model for early-stage design space exploration. In *Proceedings of the conference on Design, Automation and Test in Europe*, pages 423–428. European Design and Automation Association, 2009.

[54] Yejoong Kim, Wanyeong Jung, Inhee Lee, Qing Dong, Michael Henry, Dennis Sylvester, and David Blaauw. 27.8 a static contention-free single-phase-clocked 24t flip-flop in 45nm for low-power applications. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, pages 466–467. IEEE, 2014.

[55] Yooseong Kim and Aviral Shrivastava. Cumapz: a tool to analyze memory access patterns in cuda. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 128–133. IEEE, 2011.

[56] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[57] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[58] Hsiang-Tsung Kung. Why systolic architectures? *IEEE computer*, 15(1):37–46, 1982.

[59] Jakub Kurzak, Stanimire Tomov, and Jack Dongarra. Autotuning gemm kernels for the fermi gpu. *IEEE Transactions on Parallel and Distributed Systems*, 23(11):2045–2057, 2012.

[60] Junjie Lai. *Throughput-oriented analytical models for performance estimation on programmable hardware accelerators.* PhD thesis, Université Rennes 1, 2013.

[61] Junjie Lai and André Seznec. Performance upper bound analysis and optimization of sgemm on fermi and kepler gpus. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–10. IEEE Computer Society, 2013.

[62] Minseok Lee, Seokwoo Song, Joosik Moon, John Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. Improving gpgpu resource utilization through alternative thread block scheduling. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 260–271. IEEE, 2014.

[63] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J Smola. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 661–670. ACM, 2014.

[64] Youjie Li, Jongse Park, Mohammad Alian, Yifan Yuan, Zheng Qu, Peitian Pan, Ren Wang, Alexander Schwing, Hadi Esmaeilzadeh, and Nam Sung Kim. A network-centric hardware/algorithm co-design to accelerate distributed training of deep neural networks. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 175–188. IEEE, 2018.

[65] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature pyramid networks for object detection.

In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2117–2125, 2017.

[66] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 2980–2988, 2017.

[67] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pages 553–564. IEEE, 2017.

[68] Minh-Thang Luong and Christopher D. Manning. Achieving open vocabulary neural machine translation with hybrid word-character models. In *Association for Computational Linguistics (ACL)*, Berlin, Germany, August 2016.

[69] Sangkug Lym, Armand Behroozi, Wei Wen, Ge Li, Yongkee Kwon, and Mattan Erez. Mini-batch serialization: Cnn training with inter-layer data reuse. *arXiv preprint arXiv:1810.00307*, 2018.

[70] Xinxin Mei and Xiaowen Chu. Dissecting gpu memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86, 2017.

[71] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

[72] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient transfer learning. *CoRR, abs/1611.06440*, 2016.

[73] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[74] nvidia. Nvidia tesla p100. *White paper*, 2016.

[75] nvidia. Nvidia tesla p100 gpu architecture. *White paper*, 2016.

[76] nvidia. Nvidia tesla v100 gpu architecture. *White paper*, 2017.

[77] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 27–40. ACM, 2017.

[78] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich,

Satish Nadathur, et al. Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications. *arXiv preprint arXiv:1811.09886*, 2018.

[79] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[80] Jason Power, Joel Hestness, Marc S Orr, Mark D Hill, and David A Wood. gem5-gpu: A heterogeneous cpu-gpu simulator. *IEEE Computer Architecture Letters*, 14(1):34–36, 2014.

[81] Raul Puri, Robert Kirby, Nikolai Yakovenko, and Bryan Catanzaro. Large scale language modeling: Converging on 40gb of text in four hours. In *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 290–297. IEEE, 2018.

[82] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*, 2016.

[83] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4780–4789, 2019.

[84] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[85] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

[86] Mark Robins. The future of deep learning: Challenges & solutions. In *Proceedings of the Computing Frontiers Conference*, pages ii–ii. ACM, 2017.

[87] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018.

[88] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[89] Holger Schwenk and Jean-Luc Gauvain. Training neural network language models on very large corpora. In *Proceedings of the conference on*

Human Language Technology and Empirical Methods in Natural Language Processing, pages 201–208. Association for Computational Linguistics, 2005.

[90] Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. A performance analysis framework for identifying potential benefits in gpgpu applications. In *ACM SIGPLAN Notices*, volume 47, pages 11–22. ACM, 2012.

[91] Noah Simon and Robert Tibshirani. Standardization and the group lasso penalty. *Statistica Sinica*, 22(3):983, 2012.

[92] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[93] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.

[94] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.

[95] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, volume 4, page 12, 2017.

[96] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[97] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996.

[98] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[99] Thomas Vogelsang. Understanding the energy consumption of dynamic random access memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 363–374. IEEE Computer Society, 2010.

[100] Qiang Wang and Xiaowen Chu. Gpgpu performance estimation with core and memory frequency scaling. *arXiv preprint arXiv:1701.05308*, 2017.

[101] Xiaolong Wang, Abhinav Shrivastava, and Abhinav Gupta. A-fast-rcnn: Hard positive generation via adversary for object detection. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[102] Wei Wen, Yuxiong He, Samyam Rajbhandari, Minjia Zhang, Wenhan Wang, Fang Liu, Bin Hu, Yiran Chen, and Hai Li. Learning intrinsic sparse structures within long short-term memory. *arXiv preprint arXiv:1709.05027*, 2017.

[103] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 2074–2082. Curran Associates, Inc., 2016.

[104] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in neural information processing systems*, pages 1509–1519, 2017.

[105] Gene Wu, Joseph L Greathouse, Alexander Lyashevsky, Nuwan Jayasena, and Derek Chiou. Gpgpu performance and power estimation using machine learning. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 564–576. IEEE, 2015.

[106] Yuxin Wu and Kaiming He. Group normalization. *arXiv preprint arXiv:1803.08494*, 2018.

[107] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He.

Aggregated residual transformations for deep neural networks. In *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on*, pages 5987–5995. IEEE, 2017.

[108] Yang You, Igor Gitman, and Boris Ginsburg. Scaling sgd batch size to 32k for imagenet training. *arXiv preprint arXiv:1708.03888*, 6, 2017.

[109] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. Scalpel: Customizing dnn pruning to the underlying hardware parallelism. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 548–560. ACM, 2017.

[110] Ming Yuan and Yi Lin. Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(1):49–67, 2006.

[111] Hao Zhang, Hyuk Jae Lee, and Seok-Bum Ko. Efficient fixed/floating-point merged mixed-precision multiply-accumulate unit for deep learning processors. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2018.

[112] Ruizhe Zhao and Wayne Luk. Efficient structured pruning and architecture searching for group convolution. In *Proceedings of the IEEE International Conference on Computer Vision Workshops*, pages 0–0, 2019.

[113] Hao Zhou, Jose M Alvarez, and Fatih Porikli. Less is more: Towards compact cnns. In *European Conference on Computer Vision*, pages 662–677. Springer, 2016.

[114] Keren Zhou, Guangming Tan, Xiuxia Zhang, Chaowei Wang, and Ninghui Sun. A performance analysis framework for exploiting gpu microarchitectural capability. In *Proceedings of the International Conference on Supercomputing*, page 15. ACM, 2017.

[115] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.

Vita

Sangkug Lym received his Bachelor of Science degree in Electrical and Computer Engineering from Hanyang University in South Korea. Then, he joined the DRAM chip design division of SK hynix, Inc., where he designed phase change memory chip architecture and its core memory functional operation circuits. During his doctoral study, his research interests are focused on machine learning application acceleration, high performance memory system, and system resilience. His research has been published in multiple top-tier computer architecture and system conferences such as ISCA, HPCA, SC, and SysML.

Permanent address: sklym@utexas.edu

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.