

國立臺灣大學電機資訊學院資訊工程學系

碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

著重於記憶體子系統的深度神經網路訓練效能分析模型

A Performance Analytical Model for DNN Training with Focus on
Memory Subsystem

蔡承佑

Cheng-Yu Tsai

指導教授：楊佳玲 博士

Advisor: Chia-Lin Yang, Ph.D.

中華民國 110 年 6 月

June 2021

國立臺灣大學碩士學位論文

口試委員會審定書

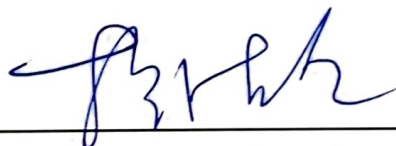
著重於記憶體子系統的深度神經網路訓練

效能分析模型

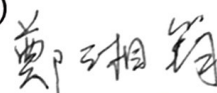
A Performance Analytical Model for DNN Training with Focus on Memory Subsystem

本論文係蔡承佑君（學號 R07922182）在國立臺灣大學資訊工程學系完成之碩士學位論文，於民國 110 年 1 月 29 日承下列考試委員審查通過及口試及格，特此證明

口試委員：



 (指導教授)



系主任



致謝

碩士班的這兩年多來，要感謝很多人的指導及陪伴。首先，最要感謝的是指導教授楊教授在這兩年多來的指導，使我在簡報表達、論述能力、思考問題及組織問題的能力等方面有很大的進步，在碩士班的這兩年多的 individual meeting 和 group meeting 從老師身上都能學到很多，像是論述問題的方式、思考問題的角度、投影片適當的呈現方式、讀 conference paper 及 paper review 時應該思考的問題等，乃至於許多做人做事的態度。除研究和論文外，也在其他領域上感受到這些年來受老師潛移默化得到的成長。老師的「high-level」的口頭禪，與李琳山教授上課常提到的「見樹不見林、見樹又見林」相互輝映，在撰寫論文的時候讓我更有方向應該要如何組織文章。研究方面同時也要感謝呂士濂博士及 Rachata 教授這兩年來對於這個計畫和論文提供的協助，在迷失方向的時刻總能獲得一些指引。也要感謝口試委員陳教授及鄭教授在口試後提供的諸多指教，點出了許多在當時並未想到的盲點，也因為這些意見讓我更加清楚的理解到自己的不足，讓這篇論文在修改後能夠組織得更好。感謝一起擔任助教的諺廷、奕青、連鎧，擔任百人大班的助教真的會遇到很多形形色色的學生，在處理作業和考試的過程也從中學到很多難得的經驗。

也要非常感謝父母在求學階段以來在物質上及心靈上持續不斷的無條件支持，讓我能衣食無虞且無後顧之憂的攻讀碩士班及做自己想做的事。每次南下回家充電總能滿電而歸，何其有幸能擁有這麼支持自己的家人。感謝電機系羽的朋友，尤其是 B04 和 B05 的老戰友們。每週四的練球時間總是最無憂無慮的時光，在球場上奔馳及揮灑汗水的時候總能忘卻許多煩惱和壓力。其中要特別感謝文彥，在這段日子裡的吃飯聊天陪我聊了很多，排解了許許多多的壓力和不快。

感謝南友會這些年來來去去的人們，這個小小社團彷彿正是個社會的縮影，存在許多形形色色的人，帶領器材組及幫忙南夜的這段日子同時也讓我自身學到了不少。謝謝積累長達四年的攝影講座及講義，讓我在研究以外的領域能夠應用在這段日子來學到的論述能力。另外要特別感謝正好陪伴我碩士班這兩年的耕宇、錫昭、映亘，無論是共事或是閒聊的時候，都能讓我感到很放心和放鬆，並且在我想要出遠門放長假的時候總能有同行的夥伴。

摘要

自從 AlexNet 在 2012 年的 ImageNet challenge 的突破後，深度神經網路 (DNN) 已經在眾多領域展現其價值。而現今許多 DNN 的硬體加速器設計都是採用小的晶片上快取 (on-chip cache) 搭配大的晶片外記憶體 (off-chip memory) 以避免頻繁的資料讀寫耗費太多時間或能量。然而，隨著科技及晶片製程的演進，除了上述的設計外，硬體設計者開始擁有更多的記憶體設計的選項。因此擁有一個用來衡量各種記憶體搭配的優劣利弊的工具變得重要。

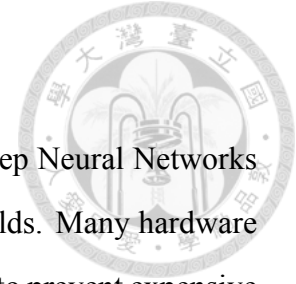
然而，現存的工具存在以下的限制：1) 只能用於推論 (inference)，不能用於神經網路的訓練 (training) 2) 只用圖像辨識的神經網路作為主要的效能評估指標 3) 只有模擬卷積層 (convolutional layer) 內部的資料流 (dataflow)，而忽略其他例如批正規層 (batch normalization layer)、活化層 (activation layer) 等層影響。我們認為神經網路的訓練對於拓展應用領域或是研究更有效率的網路結構皆極其重要，且除了卷積層及全連接層以外的層，在神經網路中訓練也具有不可忽略的影響。

在這篇論文中，我們提出了一個著重於記憶體的神經網路訓練效能分析模型。這個分析模型以神經網路結構、晶片上快取的容量、晶片外快取的頻寬作為輸入參數，假設採用幾近最佳化的軟體管理快取 (software-managed cache) 以避開快取設計中實作細節對效能的折扣，預估這組輸入參數下能夠得到的訓練效能，例如訓練一回合需要的執行時間、平均頻寬、資料搬移量等等。

這篇論文具有以下貢獻：1) 提出一個可以用於評估整個深度神經網路訓練過程效能的模型，並且有將過程中的所有層皆考慮進去，而非只考量某些計算量較大的層。2) 對於深度神經網路中各種規模的資料再利用提出徹底的分析。3) 提出幾項對於現行神經網路的觀察及建議以提供未來深度神經網路的研究及優化可著重的方向。

關鍵字：深度神經網路、神經網路訓練、頻寬、快取容量、分析模型、資料再利用

Abstract



Since the breakthrough of AlexNet in 2012 ImageNet challenge, Deep Neural Networks (DNNs) has been proving the effectiveness in various computing fields. Many hardware designs combine a small on-chip cache with large off-chip memories to prevent expensive memory access. With the progress of technology, hardware designers are having more and more design choices. A tool to estimate the tradeoffs among all memory design parameters is thus important.

However, existing tools are limited to i) inference only; ii) image classification networks as the primary benchmark for evaluation; iii) only model the dataflow within the convolutional layers, neglecting other layers like batch normalization and activation. We believe that training networks is still important to extend the applications and develop more efficient model structure, and layers except convolutional and fully-connected layers still play a non-negligible role in DNN training.

In this work, we propose an analytical model for DNN training with focus on memory. The analytical model takes the DNN model, on-chip cache capacity, off-chip memory bandwidth as input, and assumes a near-optimal software-managed cache to bypass the issue of implementation detail in the cache design, to estimate some performance metrics like execution time, average bandwidth, memory traffic, etc., of a training iteration.

This work has the following contributions: i) proposing an analytical model to estimate the performance of a whole DNN training iteration rather than some selected layers, ii) provide a thorough analysis of DNN architecture in all scales, iii) several observations and recommendations on where future DNN training research and optimization should be focused are proposed.

Key words: Deep Neural Network, training, bandwidth, cache capacity, analytical model, data reuse



Contents

| | |
|--|------------|
| 口試委員會審定書 | i |
| 致謝 | ii |
| 摘要 | iii |
| Abstract | iv |
| List of Figures | vii |
| List of Tables | ix |
| 1 Introduction | 1 |
| 2 Related Work | 4 |
| 2.1 Analytical Model | 6 |
| 2.2 Simulator | 6 |
| 2.3 Benchmarking and Profiling | 7 |
| 3 Data Reuse in ML Training | 9 |
| 3.1 Deep Neural Network Training | 9 |
| 3.2 Main Data Types | 9 |
| 3.3 Reuse Scopes | 10 |
| 3.3.1 Intra-layer Reuse | 11 |
| 3.3.2 Adjacent-layer Reuse | 11 |
| 3.3.3 Block Scale Reuse | 11 |

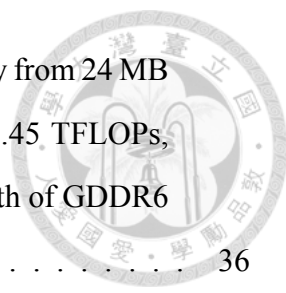


| | | |
|----------|---|-----------|
| 3.3.4 | Recurrent Weight Reuse | 12 |
| 3.3.5 | Forward-backward Reuse | 13 |
| 3.4 | Operational Intensity and Reuse Frequency | 13 |
| 4 | Methodology | 16 |
| 4.1 | Problem Definition | 16 |
| 4.2 | Framework Overview | 17 |
| 4.2.1 | Model Transformation | 19 |
| 4.2.2 | Layer execution order scheduling | 19 |
| 4.2.3 | access_list, prefetch_list construction | 19 |
| 4.2.4 | Layer-wise execution | 21 |
| 4.2.5 | Performance estimations | 21 |
| 4.3 | Challenges and Limitations of this Work | 23 |
| 4.3.1 | Hardware Platform | 24 |
| 4.3.2 | Complexity of Cache Management Policy | 24 |
| 4.3.3 | Scope of this Work | 24 |
| 5 | Experiment Results | 26 |
| 5.1 | Workloads | 26 |
| 5.2 | Characteristic of the Workloads | 27 |
| 5.3 | Overall Performance | 30 |
| 5.3.1 | Memory Traffic | 30 |
| 5.3.2 | Execution Time | 33 |
| 5.3.3 | Average Bandwidth | 36 |
| 5.4 | Layer Execution Time Breakdown | 38 |
| 5.5 | Effect of Batch Size | 39 |
| 6 | Conclusion | 42 |
| | Bibliography | 44 |



List of Figures

| | | |
|-----|--|----|
| 3.1 | Training a Deep Neural Network (DNN) | 9 |
| 3.2 | Different scope of reuse pattern | 11 |
| 3.3 | Residual block proposed in ResNet [1] | 12 |
| 3.4 | Unrolling of an RNN cell | 12 |
| 4.1 | Comparison of naive prefetch-next-layer policy and our latency-aware prefetching policy | 18 |
| 4.2 | In this example, when executing Layer 3 forward, the cache must vacate some space to accommodate the output of Layer 3. Our load-ahead of-flooding offloads the tensors of long reuse distance in advance, rather than waiting until the cache runs out of space. | 18 |
| 4.3 | The start_time and end_time of a layer | 22 |
| 5.1 | Our modified roofline model uses <i>reuse frequency</i> as the x-axis to avoid the effect of cache capacity. To have fair comparison, the ridge point in four figures are the same, which is the one of RTX 2080 Ti (Table 5.2). | 28 |
| 5.2 | Memory traffic in a training iteration with cache capacity from 24 MB to 1000 MB. | 32 |
| 5.3 | Execution time (left axis) of a training iteration with cache capacity from 24 MB to 1000 MB. The throughput of 2080 Ti is 13.45 TFLOPs, and 6900 XT is 23.04 TFLOPs. The speedup (right axis) is the execution time of 2080 Ti divided by the one of 6900 XT. The dashed line at the top is the theoretical upper bound of the speedup, which is $\frac{23.04}{13.45} = 1.71$ | 34 |



| | | |
|-----|--|----|
| 5.4 | Average bandwidth of a training iteration with cache capacity from 24 MB to 1000 MB. The throughput of both lines are set to be 13.45 TFLOPs, which is the same as RTX 2080 Ti. The maximum bandwidth of GDDR6 is 616 GB/s, and that of HBM2e is 1555 GB/s | 36 |
| 5.5 | Execution time breakdown with different cache capacity | 39 |
| 5.6 | Throughput at different batch size. The blue bars (left axis) represent the throughput when cache is 24 MB, and the orange bars (right axis) represent the throughput when cache is 400 MB. | 40 |



List of Tables

| | | |
|-----|--|----|
| 3.1 | Major data types in forward propagation and backward propagation | 10 |
| 5.1 | Workload used in this work. | 27 |
| 5.2 | Existing hardware configuration | 31 |



Chapter 1

Introduction

Since the breakthrough of AlexNet [2], there have been many machine learning models based on the DNN approach. These models are developed for different applications ranging from the original image classification to object detection, speech recognition, machine translation, natural language processing, and advertisement recommendation system. A Deep Neural Network (DNN) usually consists of tens or hundreds of layers of neurons stacked together to achieve the desired prediction task. With so many neurons, the computation demand is very high. There have been many special-purposed hardware designed to accelerate the computation demand of DNNs. With the size of neurons in hundreds of megabytes, the amount of memory needed to store model parameters and activation results is very high.

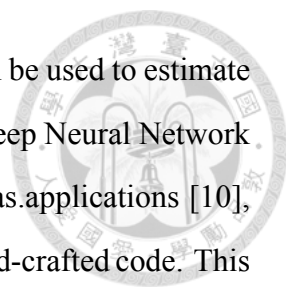
With continuing advancement of semiconductor technology, not only logic density is getting higher, on-chip memory compactness has been increasing as well. For example, an SRAM bit cell area reduces from $0.074 \mu m^2$ (in 16nm circa 2014) to $0.021 \mu m^2$ (in 5 nm circa 2020) [3]. With the reticle size limited to a bit over $800 mm^2$, balancing the amount of on-chip memory with computation logic remains to be one of the key trade-offs facing chip designers. One could put a large amount of SRAM on chip to alleviate the memory bandwidth pressure resulting from the soaring amount of computation. For example, AMD Radeon RX 6000 series GPU [4] utilizes an approach called *infinity cache* which utilizes a large on-chip memory up to 128 MB. While another design, NVIDIA RTX 3090, [5][6] has only 6 MB of SRAM for the its last-level cache (LLC). There are other

implications beyond just the area trade-off between SRAM capacity and logic. Increasing SRAM capacity may also increase on-chip memory latency and standby power.

To complicate the design trade-off of balancing between on-chip memory size and computation capability, there are several different off-chip memory options to consider. DDRx DRAM (commonly used in CPU based system) is less expensive and can provide large capacity but has a lower bandwidth. Graphic DRAM such as GDDR6 achieves higher bandwidth with higher IO frequency but has lower capacity and it is costlier and consumes more power. Newer 3D memory like High-Bandwidth Memory (HBM) supports very high bandwidth but is much more expensive. It is also harder to scale to larger capacity. With ever increasing DNN model size [7], the pressure on the off-chip memory size will continue. With the above mentioned challenges, a tool to estimate the trade-offs among on-chip computation/memory partition, off-chip memory selection with performance and cost in mind is highly desirable.

In this work, we develop an analytical model to estimate the performance of a DNN training task with memory subsystem design in mind. The *capacity of on-chip cache* affects the data reuse in the computation. With a larger cache capacity a longer reuse distance can be captured thus reducing the off-chip memory bandwidth pressure. At the same time, the *off-chip memory bandwidth* determines the time required to move the needed caching data into or out of the chip. The *computational throughput* indicates the computation time required to finish a given number of operations. If the computational throughput and off-chip memory bandwidth is perfectly matched, the computation time and memory I/O time can overlap completely eliminating wait time. These three factors are the most important hardware parameters affecting the performance of training a DNN workload. Our analytical model takes on-chip cache capacity, off-chip memory bandwidth, computational throughput, and the neural network model as input to forecast the effect of each of these parameters to the DNN performance. The goal is to appraise the benefit of a very large on-chip memory to aid potential emerging memory technology development.

Our analytical model can support all operations involved in DNNs such as an image classification [1][8] network architecture with lots of CNN layers, an object detection net-



work [9] and an RNN-based language model network. Our model can be used to estimate the cache traffic, memory bandwidth, and total execution time of a Deep Neural Network (DNN) training iteration. This tool supports DNN models from `keras.applications` [10], `darknet` [11][9], and two machine translation models [12][13] with hand-crafted code. This tool can help accelerator designers to explore the influence of cache capacity, off-chip memory bandwidth, and computational throughput to the performance of DNN training, and identify the bottleneck by layer-wise analysis and roofline model. We also provide some observations from the state-of-the-art networks.

This work makes the following three main contributions.

- We provide an analytical model for DNN training which accepts any existing networks in `keras.applications` [10] or `darknet` [11][9] as input for trade-offs study. It also allows a designer to understand execution time, memory bandwidth requirement, compute utilization, and on-chip buffer usage of each layer to spot the bottleneck for these networks in training.
- We explore all possible aspect of reuse opportunity. Previous works mainly focus on intra-layer reuse only. However, the amount of intra-layer reuse is limited. With continue scaling of SRAM and the availability of emerging memory technologies (e.g. MRAM), it is certainly possible to include a very large on-chip cache or buffer such as the one in AMD RX6900XT [4] (128 MB). It is essential to evaluate inter-layer reuse as well. Our model is able to evaluate the inter-layer reuse along the whole training iteration.
- We demonstrate the model capability by evaluating a few existing state-of-the-art DNNs and provide key observations for each with this tool. We project network training performance given various memory system parameters. These insights motivate technologists to invest research in emerging memory technologies which provide even denser cell size.



Chapter 2

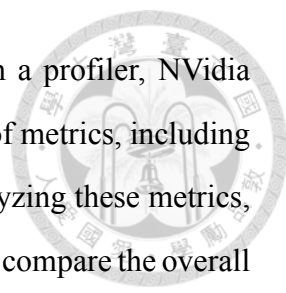
Related Work

In this chapter, we introduce works related to ours and their limitations. These works can be categorized into three major types: analytical model, simulator, and benchmarking/profiling. We will give an overview of each major types, and point out some common limitations that possessed by multiple works. Then we breakdown into three sections to introduce each works in detail.

Analytical models apply needed arithmetic formula to emulate the workload executing on a pasteurized hardware model according to known understanding. They are usually efficient and can obtain results much faster than simulation-based models. For example, MAESTRO [14] claims that they are $1029.1 - 4116.3\times$ faster than an equivalent RTL simulation. However, those formula are only applicable to specific targeted workloads.

Simulators typically run actual workloads on a software described hardware model. They more accurate and can accommodate different workloads once the hardware model is specified. Unfortunately simulation may run very slowly for DNNs with large number of layers on a software modeled hardware with many parameters. For example, GPGPU-Sim [15] runs at thousands of instructions per second, it will require days or weeks to complete a complex full DNN workload. It is unrealistic to use this approach for design space exploration (DSE).

Benchmarking selects some representative workloads in several application fields as benchmarks. Beside this, some of them run these selected workloads with *profiling* to provide observations on state-of-the-art applications, which is the part related to ours.



Profiling works run the training or inference program together with a profiler, NVidia Visual Profiler (NVVP) for example. The profiler records some sort of metrics, including average bandwidth, compute utilization, etc., at each kernel. By analyzing these metrics, researchers can figure out the bottleneck of a workload. They may also compare the overall performance with various batch size or DL framework. The shortcoming of these works is that they only work for existing hardware. They cannot be used to predict the performance on non-existing hardware configuration, for example, a much larger cache.

Then we introduce some common design choices and the incurred limitations that multiple works have.

Limitation 1: Modeling the dataflow within computation-intensive layers like convolution and fully-connected. The interactions between layers are neglected, and some point-wise layers like batch normalization and ReLU are not considered. Some works focus on the cycle-level dataflow within computation-intensive layers, like loop orders and layer tiling techniques. They discuss the most efficient way to divide a bunch of tensors and the best way to pass them into PEs. Though we also agree that this topic is still important; however, these works cannot be extended to large cache, in which caching more than one layer is totally possible. These works cannot model the interactions between layers, and their focuses are only on computation-intensive layers, none of them models the point-wise layers, which exist in almost all DNNs and take non-negligible portion of execution time. Specifically, TBD [16] observed that 26.04% of the execution time is consumed by point-wise layers in training ResNet on Quadra P4000. Those point-wise layers should not be neglected while estimating the end-to-end execution time.

Limitation 2: Only supports forward propagation. Back propagation is not considered. As we will introduce in 3.3, in terms of data reuse, back propagation requires a very long reuse distance, which does not exist in inference. This forward-backward reuse causes a huge difference. Besides the data reuse, the number of operations in back propagation is approximately $2\times$ as many as those in forward propagation. Moreover, as we will show in Table 3.1, the data requirements in forward and backward propagation are also different. It is not just a matter of multiplying the execution time by 2 or 3 to calibrate

from inference to training. The problem is far more complex than that.

Limitation 3: Only considers CNN, especially image-classification models, neglecting DNNs in other application fields. The application that gained the most focuses for designing and evaluating DNN accelerators is image classification. Possibly one of the reasons is the popularity of ImageNet Challenge. However, there are far more applications than image classification or even CNNs in deep learning. For example, RNNs and CNNs are vastly different in layers they use, network structure, feature map size, number of operations, etc. Those only model CNNs can only model very limited applications.

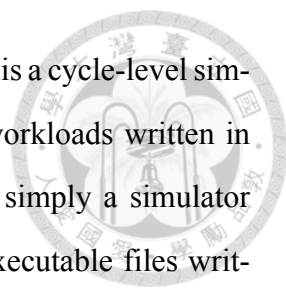
Next we will break down the related works into three sections and go through them one by one in detail.

2.1 Analytical Model

MAESTRO [14] (**M**odeling **A**ccelerator **E**fficiency via **S**patio-**T**emporal **R**esource **O**ccupancy) is an analytical cost model that takes as input 1) a DNN model with a set of layers, 2) a dataflow description for each layer, and 3) the hardware configuration. Based on these inputs, MAESTRO outputs estimates of end-to-end execution time, energy (including all compute, buffer, and interconnect activities), NoC costs, and so on. The main focus of MAESTRO is on discussing the effect of dataflow such as loop orders and loop tiling in convolutional and fully-connected layers, thus having the Limitation 1 and 3 we mentioned above.

2.2 Simulator

SCALE Sim [17] (**S**ystolic **C**NN **A**ccelerator **S**imulator) is a CNN accelerator simulator that provides estimates on cycle-accurate timing, power/energy, memory bandwidth, and trace results for a specified accelerator configuration and neural network architecture. They focus on exploring the intra-layer dataflow in convolutional and fully-connected layers on systolic array, and also having the Limitation 1 and 3 as MAESTRO [14].



GPGPU-Sim [15], a long-lasting general-purpose GPU simulator, is a cycle-level simulator modeling contemporary GPUs running on GPU computing workloads written in CUDA or OpenCL. Unlike other works mentioned above, itself is simply a simulator without DNN parser or DL framework support. It can only takes executable files written in CUDA (.cu files) or OpenCL (.cl files), but the most popular DL frameworks used by domain researchers are all Python based (.py files), e.g. TensorFlow and PyTorch. DNNs in these frameworks cannot be directly taken by GPGPU-Sim unfortunately. To our best knowledge, there are only two existing packages that can run on GPGPU-Sim [15] directly, DeepBench [18] and Tango [19]. DeepBench [18] uses the neural network libraries to benchmark the performance of basic operations on different hardware. It does not work with deep learning frameworks or deep learning models built for applications. It is used to benchmark the underlying operations involved in DNNs, cannot be used to measure the time required to train an entire model. So DeepBench has the Limitaiton 1. On the other hand, Tango [19], a DNN benchmark suite providing a set of widely used CNNs and RNNs written in CUDA C and OpenCL, unfortunately supports inference only, as we mentioned in Limitation 2. Moreover, because their neural networks are written in low-level languages, it is difficult for domain researchers to extend their works for other purposes.

2.3 Benchmarking and Profiling

TBD [16] (**T**rainin**B**enchmark for **D**NNs), is a benchmark for DNN training that uses a representative set of DNN models covering a broad range of machine learning applications: image classification, machine translation, speech recognition, adversarial networks, and reinforcement learning. TBD also incorporates an analysis tool chain for performing detailed resource and performance profiling of these models. They perform a detailed performance analysis on how these different applications behave on three DNN training frameworks (TensorFlow, MXNet, CNTK) across different hardware configurations (TITAN Xp, Quadra P4000, Xeon E5-2680) and gain some interesting insights. However, we cannot predict the performance on newer machine with vastly different hardware con-

figurations from their analysis.

Siu et. al. [20] investigate the on-chip memory requirements of state-of-the-art CNNs with the goal of minimizing off-chip traffic whilst maintaining a reasonable on-chip memory capacity. They estimate the cache capacity and bandwidth requirement under four selected caching scheme, and only consider convolutional layers in image-based applications. Thus they are having the Limitation 1 and 3.

Tango [19], as a benchmark suite written in CUDA C and OpenCL, which is able to run on GPGPU-Sim [15], tested their suite on some real machines and GPGPU-Sim and provided some observations and insights from the results. They run their workloads on NVIDIA GK210, NVIDIA TX1, GPGPU-Sim, and PynQ-Z1, with 5 CNNs (AlexNet, ResNet, SqueezeNet, CifarNet, and VGGNet) and 2 RNNs (GRU and LSTM). However, despite including results on simulators, they still have the Limitation 2.



Chapter 3

Data Reuse in ML Training

3.1 Deep Neural Network Training

Artificial Neural Networks (ANNs), a machine learning technique, take data samples as input and generates desired outputs for a particular purpose. Deep Neural Networks (DNNs) are types of ANN with multiple layers (e.g. convolution, fully-connected, pooling, batch normalization, activation, etc) between inputs and outputs. Each layer has its own set of *weights*, and applies some mathematical transformation to its input (\mathbf{x}) and weights (W) to produce the results for the downstream layers. The intermediate results are called *feature maps*, and marked as \mathbf{x} and \mathbf{y} in Figure 3.1.

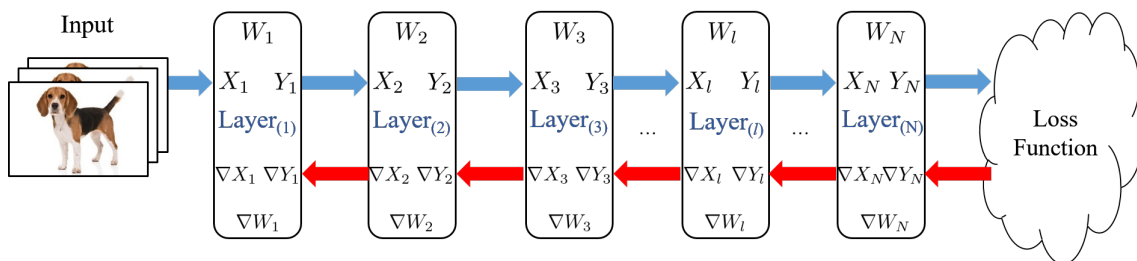
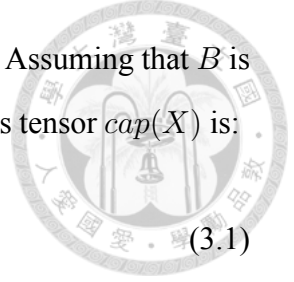


Figure 3.1: Training a Deep Neural Network (DNN)

3.2 Main Data Types

Tensors are multi-dimensional arrays of a program. For example, input samples of an image classification network is a 4-dimensional array including batch size (N), height



(H), width (W), and number of channels (C) (e.g. R, G, B channels). Assuming that B is the number of bytes of each data point in the array, the total size of this tensor $cap(X)$ is:

$$cap(X) = B \times N \times H \times W \times C \quad (3.1)$$

In forward propagation, the operation can be modeled as $Y = f(X, W)$, where X is the input feature(s), W is trainable model parameters, and Y is the output feature of a layer. Some non-trainable layers (e.g. ReLU) may not have W , and some layers may contain more than one input tensors (e.g. point-wise add). In backward propagation, the goal is to calculate ∇W of all trainable layers. This is done through the gradient descend approach by calculating ∇X backward. More specifically, $\nabla X = f(\nabla Y, W)$, and $\nabla W = f(\nabla Y, X)$ in most cases¹.

| | tensor | input/output | last access | next access |
|----------|------------|--------------|-------------|----------------------------------|
| forward | X | input | last layer | 1) backprop 2) residual block |
| | W | input | last batch | backprop |
| | Y | output | N/A | next layer |
| backward | X | input | forward | None |
| | ∇Y | input | last layer | None |
| | W | input | forward | next batch |
| | ∇X | output | N/A | next layer |
| | ∇W | output | N/A | None |

Table 3.1: Major data types in forward propagation and backward propagation

In summary, there are 6 major data types to each layers, X , W , Y , ∇X , ∇W , and ∇Y . Table 3.1 summarizes the tensors relating to a layer in forward propagation and backward propagation respectively.

3.3 Reuse Scopes

In training a DNN, there are five different types of data reuse: intra-layer reuse, adjacent layer reuse, block-scale reuse, recurrent weight reuse, and forward-backward reuse.

¹Some special layers like tanh, softmax, ReLU layers have $\nabla X = f(\nabla Y, Y)$.

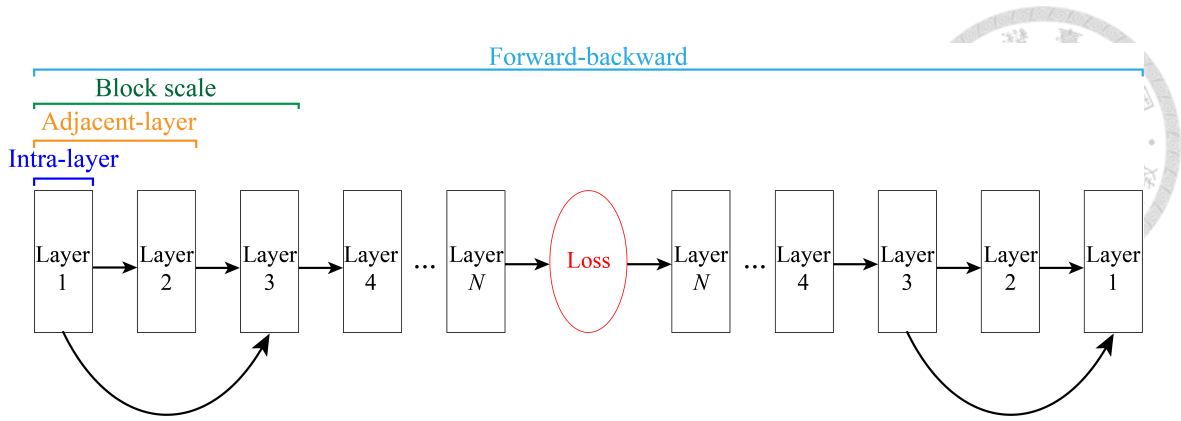


Figure 3.2: Different scope of reuse pattern

3.3.1 Intra-layer Reuse

This is a data reused within a layer of the network. The type of layer affects intra-layer reuse mostly. For example, in convolution layers, each data points in the feature maps or weight are reused up to 10^3 times. However, feature maps in point-wise layers like batchnorm, ReLU are only reused one or two times.

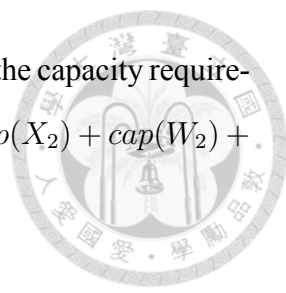
3.3.2 Adjacent-layer Reuse

This is the reuse of data within adjacent layers. For example, in Fig. 3.1, Y_2 , output of Layer 2, is essentially the same tensor as X_3 , thus is reused by Layer 3. Hence, if the output feature map of the previous layer can be kept in cache, there is no need to re-fetch. This reuse has been studied by [21]. Referring to Table 3.1, the capacity requirement for this type of data reuse is the tensors of the whole layer. During forward pass, the capacity is: $cap(X) + cap(W) + cap(Y)$, while it is $cap(X) + cap(\nabla Y) + cap(W) + cap(\nabla W) + cap(\nabla X)$ for backward pass.

Adjacent layer reuse is important for light-weighted layers such as batchnorm and ReLU because the computation time is much shorter than the off-chip data transfer time. With adjacent-layer reuse, these light-weighted layers can start directly without stall.

3.3.3 Block Scale Reuse

As Figure 3.3 shows, the residual connection proposed in ResNet [1] has been widely applied in many modern DNNs to prevent gradient vanishing. Specifically, x_1 is not only



used by layer 1 but also by the add layer. In the example of Figure 3.3, the capacity requirement to capture this residual block reuse at forward propagation is $cap(X_2) + cap(W_2) + cap(X_3) + cap(X_1)$, which is a bit larger than caching adjacent layer.

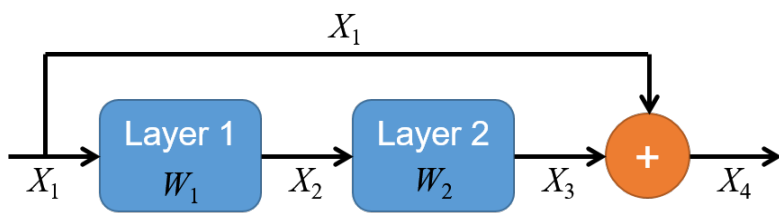


Figure 3.3: Residual block proposed in ResNet [1]

3.3.4 Recurrent Weight Reuse

The recurrent weight reuse only occurs in recurrent neural networks (RNN). Figure 3.4 shows the unrolling of an RNN cell. The weight A is repeatedly reused T times to compute the feature maps of T timesteps (h_1 to h_T). Moreover, an RNN cell is itself a small network, consisting of several layers inside. For example, an LSTM cell contains 4 fully-connected, 4 sigmoid, 2 tanh, 3 point-wise multiplication and 1 addition. Therefore, the reuse distance of weights A within the RNN cell crosses several layers. Specifically, taking Fig. 3.4 as an example, the capacity requirement for recurrent weight reuse is $cap(x_t) + cap(h_t) + cap(A)$ in forward pass, where A includes all the FC weights and intermediate features in the RNN cell. Approximately, the major part of $cap(A)$ in LSTM is the weights of the 4 FC layers inside.

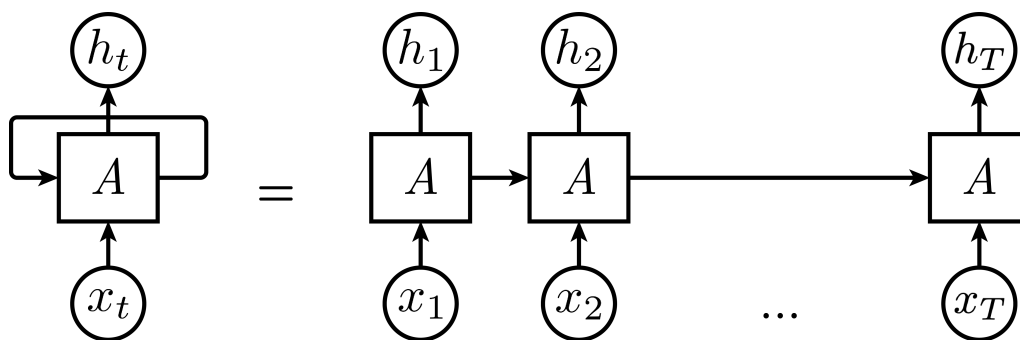
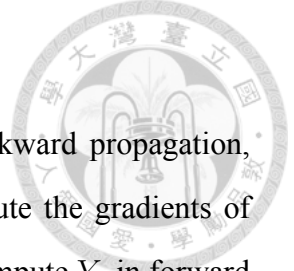


Figure 3.4: Unrolling of an RNN cell



3.3.5 Forward-backward Reuse

The main difference between training and inference is the backward propagation, which exploits the feature maps obtained at forward pass to compute the gradients of weights and feature maps. For example, in Fig. 3.1, Y_2 is used to compute Y_3 in forward pass, and reused again in backward propagation to compute ∇X_2 and ∇W_2 .

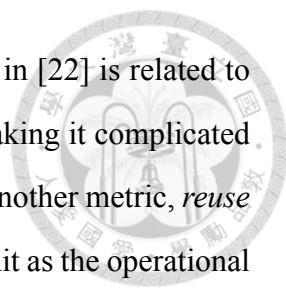
The capacity requirement to fully capture forward-backward reuse is approximately $\sum_{i=1}^N cap(Y_i) + cap(W_i)$. This value is often too large for on-chip buffering. Nevertheless, partial reuse is still possible. As Fig. 3.1 shows, in backward propagation (the red arrow), layers are executed in opposite order to the forward propagation (the blue arrow). Hence, those which are closer to the loss function have shorter reuse distance than those closer to the input, and more likely to retain in cache to be reused.

3.4 Operational Intensity and Reuse Frequency

We introduce three terms, *operational intensity*, *ridge point*, and *reuse frequency*, which will be extensively used in our further analysis.

The roofline model [22] is a visualization method that relates the processor performance and the memory bandwidth. This 2-D plane plots *operational intensity* (*operations/bytes*) as x-axis and attainable performance (FLOP/s) as y-axis. The *operational intensity*, is defined as operations per byte of DRAM traffic while computing a kernel in [22]. The *ridge point*, where the diagonal line (memory-bound) and the horizontal line (compute-bound) meets in the roofline model, is the point of peak computational performance and peak memory bandwidth, whose x-coordinate is the minimum operational intensity required to achieve maximum performance. Unless specified, we will use *ridge point* to represent the x-coordinate of this point in the following context. This minimum operational intensity is derived by Equation 3.2, where both the numerator and denominator are the hardware parameters.

$$ridge_point(\text{FLOP/bytes}) = \frac{\text{throughput (FLOP/s)}}{\text{off-chip bandwidth (bytes/s)}} \quad (3.2)$$



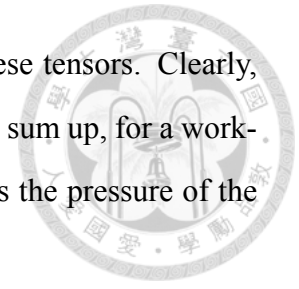
However, the off-chip traffic term to derive operational intensity in [22] is related to the cache design, and can only be obtained by running the workload, making it complicated for static analysis. To simplify the problem, we would like to define another metric, *reuse frequency* in Equation 3.3 and Equation 3.4, which shares the same unit as the operational intensity in [22]. Equation 3.3 is used to describe the reuse frequency of a tensor t in a layer l , and Equation 3.4 is for a layer l . Note that in Equation 3.3 and 3.4, the tensor capacity term is not bound to the cache design like the one in operational intensity. Furthermore, as Equation 3.3 shows, this definition can be transformed into operations per data points in the multi-dimensional array with the constant B , number of bytes per data points. This is the reason why we named it. Similar as that in [22], we describe those whose reuse frequency is larger than the ridge point as *compute-bound*, and those whose reuse frequency is lower than the ridge point as *memory-bound*.

$$rf(t, l) = \frac{operations(l, t)}{cap(t)} = \frac{operations(l, t)}{\#data_points \times B} \quad (3.3)$$

$$rf(l) = \frac{operations(l)}{\sum_{t \in l} cap(t)} \quad (3.4)$$

Although replacing the off-chip traffic with the tensor capacity may not precisely present what it really happens, what we want to do by defining this term is to quickly evaluate whether a workload is tend to be compute-bound or memory-bound like the idea of [22] without considering the cache design. To be more specific, when compute-bound tensors are being computed, it is likely that more time is spent on computing them than transferring them between cache and DRAM. Therefore, while computing compute-bound tensors, it is possible to prefetch or offload other tensors at the same time to hide the data transfer latency. However, while the processing tensors are memory-bound, the time required to load the tensors from off-chip is longer than the time they spend on computation, which may harm the compute utilization. In this way, ideas to improve the utilization when reaching these tensors are 1) prefetching them to overlap the latency of loading them with computation time of compute-bound tensors 2) having data reuse to prevent data transfer

3) increasing off-chip bandwidth to decrease the time to transfer these tensors. Clearly, there must be enough cache capacity for solution 1 and 2 to work. To sum up, for a workload, the proportion of memory-bound tensors can be an indicator as the pressure of the memory system.





Chapter 4

Methodology

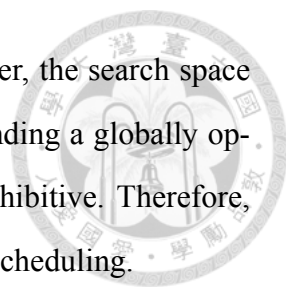
In this section, we give a high-level view of our framework. We first define the problem we aim to solve, and then present an overview of our framework. We will also analyze the challenges of the problem.

4.1 Problem Definition

The problem we aim to solve in this work is defined as follows. Given the capacity of the on-chip cache, bandwidth to the off-chip memory, the throughput of the computation units, and the neural network configuration. **Our goal** is to estimate the performance and its related metrics (e.g., total execution time, average computation utilization, average off-chip bandwidth, and total off-chip memory traffic) in a training iteration under the management policy that make the best use of the cache such that the stall time and the memory traffic is minimized.

To obtain the estimates on the execution time and average bandwidth, we use number of operations divided by throughput to estimate the compute time, and the number of bytes divided by off-chip bandwidth to estimate the data transfer time. We make no assumption on the underlying hardware platform and optimistically assumes the accelerator and the memory bus can achieve full utilization.

We assume an optimal software-managed cache, whose stall time and off-chip memory traffic can be minimized. We believe that only by applying the optimal data transfer



scheduling can we discuss the ultimate hardware limitation. However, the search space for optimally scheduling the data transfer is exponentially large. Finding a globally optimal solution for the scheduling problem seems computationally prohibitive. Therefore, we proposed some heuristics to effectively approximate the optimal scheduling.

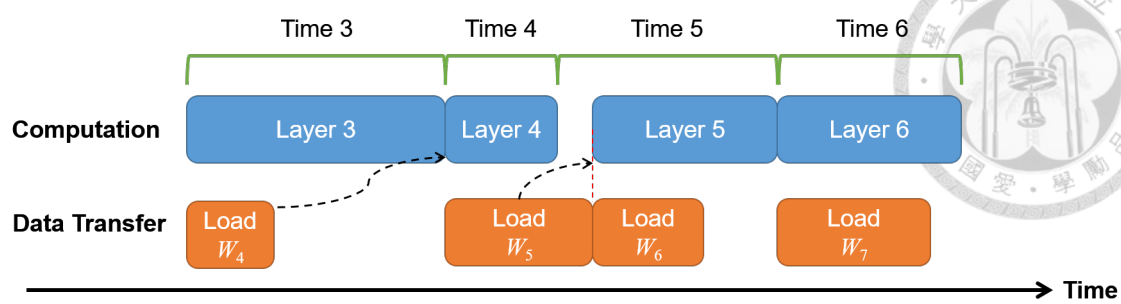
First, The *Bélády's algorithm* is taken as our cache replacement policy, which has been proved to be optimal replacement policy, exploiting the network information to replace the tensor whose next use is the farthest in the future.

Second, as Figure 4.1 shows, our prefetching scheduling takes the data transfer latency into consideration. For tensor required at layer i , it will be scheduled at layer $i - k$ so that the computation time from $i - k$ to $i - 1$ is larger than the loading latency to assure the latency can be overlapped with computation time.

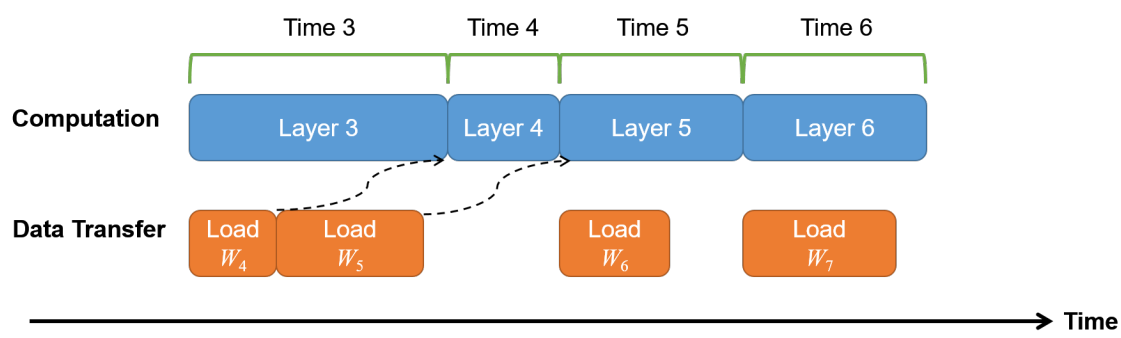
Third, for the offloading part, our cache performs a prediction mechanism to look ahead whether a tensor can be kept in cache until its next use under the Bélády's replacement policy. If a tensor is known in advance that cannot be kept in cache until its next use, it will be offloaded immediately after the current use. As Figure 4.2 shows, if the offloading scheduling relies solely on the Bélády's replacement policy, there is still inevitable stall caused by the offloading latency when computing Layer 3 at forward propagation (Time 3f). With our look-ahead offloading scheme, X_1 , the tensor of Layer 1, is offloaded immediately after forward propagation of Layer 1 is done because the cache cannot keep it until its next use.

4.2 Framework Overview

Listing 1 shows the complete flow of our framework. The flow can be roughly broken down into 5 steps. 1) The neural network in keras.applications [10] format first go through *model transformation* and parsing to be fed into our model. 2) We apply a recursive algorithm in [23] to *schedule the execution order* of the layers in the network. 3) *estimates the prefetching timing and record the access pattern* according to the execution order and hardware configuration. 4) In the *layer-wise execution*, we iteratively offload unused tensors, prefetch future tensors, and allocate space for output tensor of the current

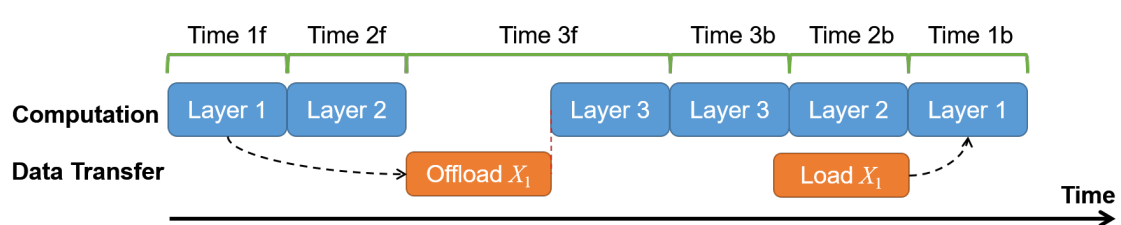


(a) prefetch-next-layer policy: delay is caused at the beginning of Layer 5 due to unmatched computation and data transfer time

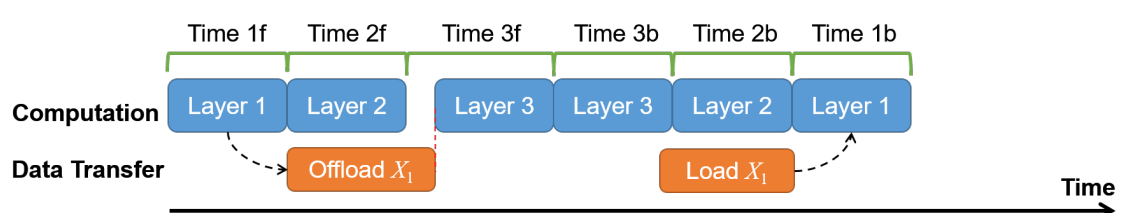


(b) latency-aware prefetching: the latency to prefetching tensors is considered during scheduling the prefetching, so W_5 is scheduled to be prefetched at the computation-intensive layer Layer 3.

Figure 4.1: Comparison of naive prefetch-next-layer policy and our latency-aware prefetching policy



(a) without look-ahead: If the tensor is only offloaded when the cache is full, there will be a delay when the cache is full.



(b) look-ahead offloading: if X_1 is known to be unable to stay in the cache until back propagation of Layer 1, it should be offloaded immediate after the forward propagation of Layer 1.

Figure 4.2: In this example, when executing Layer 3 forward, the cache must vacate some space to accommodate the output of Layer 3. Our load-ahead offloading offloads the tensors of long reuse distance in advance, rather than waiting until the cache runs out of space.

layer. 5) We estimate the performance for each single layers and the overall performance. We introduce the details of each steps in the following subsections.



4.2.1 Model Transformation

Our model can take DNNs in `keras.applications` [10] as input. The DNN will undergo a model transformation which dumps the selected DNN into a network description plain text file. Then a parser parse this text file into the format that can be processed internally (line 1). The network description file consists of four elements for each layers: layer name, layer type, input layer(s), and the parameters required to estimate the output feature map size and number of operations.

4.2.2 Layer execution order scheduling

After the DNN model is read, the first step before anything can start is to schedule the layer execution order (line 2). We use the algorithm proposed by [23], which recursively explores the subsequent layers in Depth-First Searching (DFS), except that it reaches a join where all prior layers must finish before proceeding.

4.2.3 access_list, prefetch_list construction

After the layer execution order is scheduled, line 3-14 iterate over these layers to produce `access_list` and `prefetch_list`. `access_list` is a two-dimensional list recording the tensors being accessed at certain layer. Line 5-6 collect all the tensors that will be accessed by each layers into the corresponding `access_list`. This list will be used by the cache to estimate the access pattern of the processor and to apply Bélády's replacement policy.

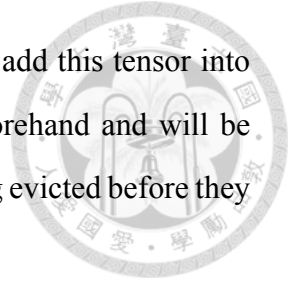
As Fig. 4.1 shows, our cache applies a *latency-aware prefetching* to assure better timing for prefetching. Line 7-14 finds the timing to prefetch and fill the results into the `prefetch_list`. For tensor that is required by i -th layer, k is the minimum number such that the total computation time of layer $i - 1, i - 2, \dots, i - k$ is no less than the



```
1 network = parse_network("nn.txt");
2 network.layers = route_construct(network);
3 # create access_list and prefetch_list:
4 for layer in network.layers:
5     for tensor in layer.access_tensors:
6         access_list[layer].push_back(tensor);
7     prev = layer.previous_layer;
8     for tensor in layer.input_tensors:
9         load_time = tensor.capacity / bandwidth;
10        accumulated_time = 0;
11        while accumulated_time < load_time:
12            accumulated_time += prev.operations / throughput;
13            prev = prev.previous_layer;
14            prefetch_list[prev].push_back(tensor);
15 # layer-wise execution:
16 for layer in network.layers:
17     # offload_tensors:
18     for tensor in cache:
19         if tensor cannot be kept in cache until next use:
20             cache.offload(tensor);
21     # prefetch_tensors:
22     for tensor in prefetch_list[layer]:
23         if (capacity enough && tensor can be kept in cache until
24             ↪ next use):
25             cache.prefetch(tensor);
26         else:
27             add tensor to the prefetch_list of the next layer
28     # compute:
29     for tensor in layer.input_tensors:
30         check tensor is in the cache;
31         otherwise: cache.load(tensor); # memory stall happens
32         ↪ here
33     cache.generate(output_tensor);
34     print_record(); # per-layer performance
35     print_statistics(); # overall performance
```

Listing 1: Execution flow of our framework

time required to load this tensor. As long as the k is found, we will add this tensor into the `prefetch_list` of the layer $i - k$. This list is created beforehand and will be adjusted during layer-wise execution to avoid prefetched tensors being evicted before they are actually used.



4.2.4 Layer-wise execution

Line 15-31 corresponds to the layer-wise execution, mainly consisting of three major steps, *offload*, *prefetch*, and *compute*. The *offload* step (line 17-20) examines all the tensors in the cache, and offload those that cannot be kept until their next use as we mentioned in Fig. 4.2. The *prefetch* step (line 21-26) tries to prefetch the tensors according to the `prefetch_list` constructed in the last paragraph. However, considering the prefetched tensors occupy the cache space and may cause the executing layer have insufficient space for storing output, an evaluation is performed before prefetching the tensors (line 23). If the tensor cannot be kept in the cache until its next use, the prefetching will be defer to the next layer.

Last, the *compute* step (line 27-31) first check the necessary inputs have been loaded into the cache; otherwise the processor have to wait for the data being loaded into the cache. Then line 31 allocates the space for output tensor of the layer.

4.2.5 Performance estimations

After the computation and data transfer is modeled, we can estimate the performance from collecting the records kept in layer-wise execution. Line 31 estimates the performance at each single layer, and line 32 estimates the overall performance throughout a training iteration. We introduce the details of each metrics in the follow paragraphs.

Memory Traffic

We keep records of each data transfers, and these records include the duration and the transferred tensor name. From these records, the memory traffic at a given range of time, for example throughout the execution of a layer, can be tracked. Memory traffic is the

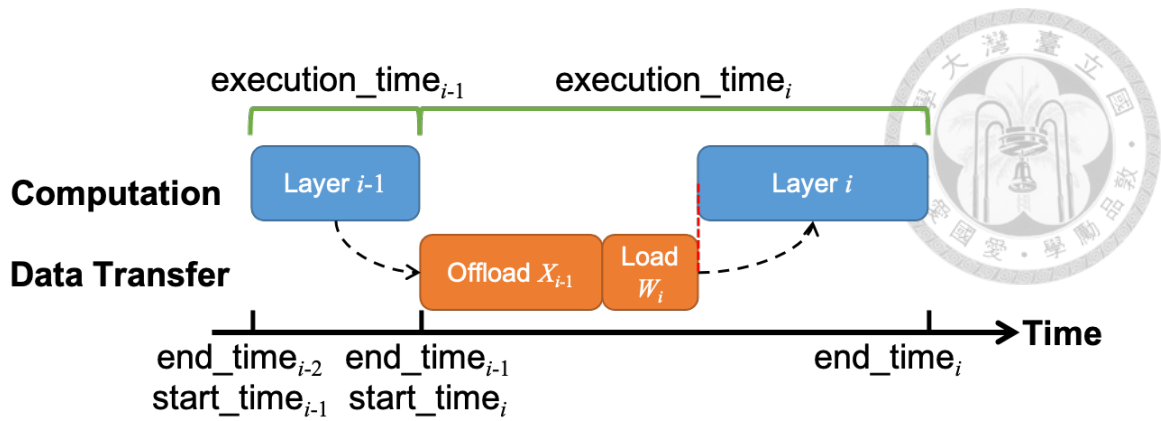


Figure 4.3: The start_time and end_time of a layer

most direct indicator of the data reuse. The more memory traffic is, the less data reuse is captured. Therefore, memory traffic is closely related to cache capacity. The larger cache is, the more data reuse it can capture, thus having less memory traffic.

Execution Time

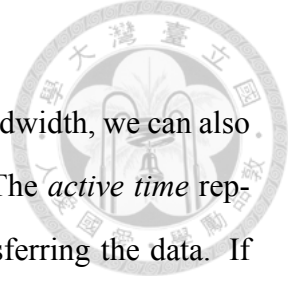
The execution time of a layer includes loading the input into the cache and computing the output. Fig. 4.3 shows an example of estimating the execution time of a single layer. We record the start_time and end_time of each layers, and the difference between them is the execution time of the layer. We estimate the compute time using the computation throughput, and estimate the data transfer time using the off-chip bandwidth.

As Fig. 4.3 shows, if there is perfect prefetching or data reuse (like layer $i - 1$), the computation can directly start without stall; otherwise, the computation units must stall until the data have been brought to cache (like layer i).

Having the execution time of each single layers, summing them up provides the total execution time of a whole iteration, which indicates the overall performance of the given hardware parameters. Another usage is that by grouping the layer of the same type, Fig. 5.5 indicates the execution time breakdown of layers of different types.

Average Bandwidth

As Equation 4.1 shows, the average bandwidth is derived from the execution time (5.3.2) and memory traffic (5.3.1) introduced above. Similarly, the observed duration can



be a single layer or a whole iteration.

Furthermore, combining the average bandwidth with the peak bandwidth, we can also estimate the *active time* of the memory system with Equation 4.2. The *active time* represents the percentage of time that the memory system is busy transferring the data. If the average bandwidth come out to be close to the peak bandwidth, the active time will be high, which means that most of the time is spent transferring the data. The compute utilization is likely to be very low due to frequent memory stalls.

$$\text{bandwidth} = \frac{\text{total bytes loaded or offloaded}}{\text{execution time}} \quad (4.1)$$

$$\text{active time} = \frac{\text{average bandwidth}}{\text{peak bandwidth}} \quad (4.2)$$

Average Compute Utilization

As Equation 4.3 shows, we estimate the compute utilization by number of operations and execution time mentioned in 5.3.2. The number of operations of each layers can be estimated from layer type and tensor size with domain knowledge.

$$\text{compute utilization} = \frac{\text{number of operations}}{\text{peak FLOP/s} \times \text{execution time}} \quad (4.3)$$

This metric represents the percentage of time that the processor is busy with computation. To be more specific, if the I/O latency cannot be perfectly overlapped with computation, the utilization drops because the processor needs to stall waiting for data at some moments. So the average utilization can provide a quick view about to what extent the computation is affected by data transfer.

4.3 Challenges and Limitations of this Work

In this section we describe the challenges of the problem we aim to solve. And we describe the assumption we made and the limitations of our work.



4.3.1 Hardware Platform

As we mentioned in 4.1, we use maximum computational throughput and off-chip bandwidth as the proxy for time estimation as [20] did. We made this assumption by two reasons. First, our work mainly focus on the memory subsystem. In order to make the modeling be generally applicable to as many platform as possible, we made no assumption on any specific hardware platform, and decided to pick the throughput, the most common parameter among all platforms, as the input to our model, instead of PE count, number of SM, etc. Second, our goal is to explore the ultimate hardware limitation, which means that the hardware design and software optimization have been made as good as possible so that the performance is only bounded by the cache capacity and off-chip bandwidth. To this end, we must assume the hardware can be almost optimal.

By the above reasons, we only take computational throughput and off-chip bandwidth as our input to estimate time, which in essence optimistically assumes the computation units and memory bus can always achieve full utilization as needed.

4.3.2 Complexity of Cache Management Policy

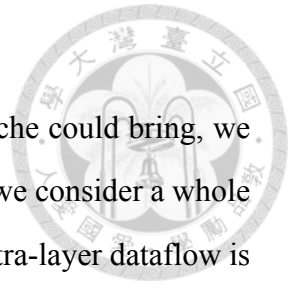
Although we would like to assume an optimal cache to explore the ultimate hardware limitation, the search space of finding a globally optimal scheduling is exponentially large [24]. For the sake of efficiency, we use greedy algorithm in 4.2.3 and 4.2.4 to approximate the optimal cache management policy.

4.3.3 Scope of this Work

As we introduced in 3.3, there are several scopes of data reuse in DNN training. It is difficult to make a model which is general enough to cover all scope of reuse but still keeps efficiency. For example, although GPGPU-Sim [15], as a general-purposed simulator, can almost model all scopes of reuse, it requires 50 hours to run only a forward propagation of ResNet-50. MAESTRO [14], as an analytical model, can run much faster, but MAESTRO is made to model intra-layer dataflow. It cannot be extended to model the inter-layer

dataflow.

Because our goal is to explore the benefit that making a large cache could bring, we must consider inter-layer dataflow under such situation. To this end, we consider a whole tensor as the basic unit to ensure efficiency. Modeling the intricate intra-layer dataflow is out of our capability. So the cache capacity we can model cannot be arbitrarily small. The minimal caching scheme that we can model is to cache a row of feature map and a whole weight, which ensures no re-fetching required to compute a layer [20].





Chapter 5

Experiment Results


Our analytical model has four input parameters: cache capacity, throughput, bandwidth, and DNN. We will introduce how we select the DNN in 5.1, and demonstrate their characteristics in 5.2. Then we showed some existing hardware configurations in Table 5.2. Unless specified, all of the following experiments are done with the configuration of RTX 2080 Ti [25].

5.1 Workloads

For image classification DNN models, our framework takes networks in `keras.applications` [10], along with a model transformation into the format that can be read by this program. However, this official support package only includes image classification DNNs. Other applications, even written in `keras`, have very different data structure from those in `keras.applications`, making it difficult to parse. So for GNMT [13] and Transformer [12], we use the hand-craft code that can be directly read into this analytical model.

ResNet-50 [1], GNMT [13], Transformer [12] are all models included in MLPerf Training [26], while MobileNet [8] is another image classification model included in MLPerf inference [27] that use more light-weighted convolutions to replace the vanilla convolution in ResNet. Note that the major difference between ResNet and MobileNet is the convolution they use. The capacity of each single tensors in these networks is still approximately the same. And the major difference between GNMT and Transformer is

the recurrent structure. Table 5.1 lists the property of these networks and the input size we set.



| Model name | Application | Dominant layer | input size |
|------------------|-------------------------------------|---------------------------------|---|
| ResNet-50 [1] | Image classification (heavy) | Convolution | $N = 32$ $H = 224$ $W = 224$ $C = 3$ |
| MobileNet v2 [8] | Image classification (light) | Depthwise Separable Convolution | $N = 32$ $H = 224$ $W = 224$ $C = 3$ |
| GNMT [13] | Machine translation (recurrent) | LSTM | $N = 32$ $H = 1024$ $T = 50$ |
| Transformer [12] | Machine translation (non-recurrent) | Multi-head Attention | $N = 128$ $H = 1024$ $T = 50$ |

Table 5.1: Workload used in this work.

5.2 Characteristic of the Workloads

In this section we are going to demonstrate the properties of each workload before including any hardware factors. To fairly compare these workloads without any hardware effect, we use a modified roofline model as shown in Fig. 5.1 to show the properties of four workloads in Table 5.1. The x-axis of Fig. 5.1 is the *reuse frequency* of each layer as we defined in 3.4, and the y-axis is the attainable performance derived from Equation 5.1, where $rf(l)$ is the reuse frequency of layer l , bw is the hardware off-chip bandwidth, and *throughput* is the computational throughput. Equation 5.1 is similar to the original definition in [22] except the reuse frequency term. To have fair comparison, the bandwidth and throughput in Fig. 5.1 are set to be the same among four workloads. In fact, the bandwidth and throughput only affect the y coordinate of the points. The x coordinate of the points are decided by the workload. So choosing the throughput and bandwidth do not affect the results in Fig. 5.1.

$$\text{attainable performance}(l) = \min\{rf(l) \times bw, \text{throughput}\} \quad (5.1)$$

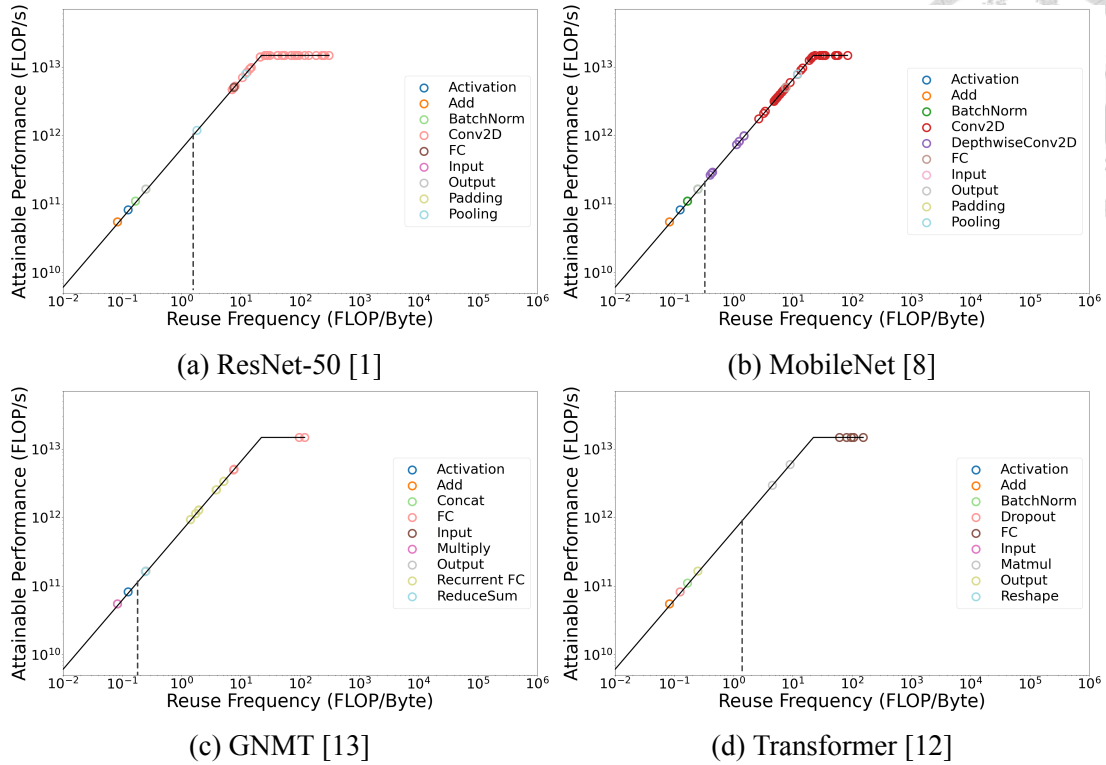
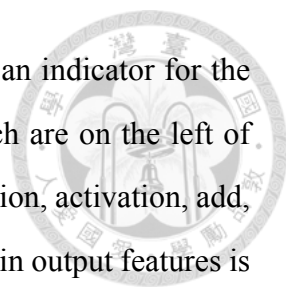


Figure 5.1: Our modified roofline model uses *reuse frequency* as the x-axis to avoid the effect of cache capacity. To have fair comparison, the ridge point in four figures are the same, which is the one of RTX 2080 Ti (Table 5.2).

Observation 1: Reuse frequency of tensors varies vastly among layers. As Fig. 5.1 shows, even in the same workload, the reuse frequency of layers can vary up to $10^2 - 10^4 \times$. Among four figures, in Fig. 5.1d, the most extreme case, these circles can even be clearly divided into two clusters. In fact, combining with our analysis to various layer types, we think that DNN layers can be categorized into two types according to their computation pattern or reuse frequency. We can categorize all layers into two types according to the definition in the next paragraph, and this taxonomy will be extensively used to explain the other observations. To illustrate this concept, we manually added a dashed line onto each figure in Fig. 5.1 to visually split the layers into two clusters. The coordinate of this dashed line is also related to the workload characteristic.

The *type-I layers*, which are on the right of the dashed line in Fig. 5.1, are feature extraction layers like convolution or fully-connected layers. In these layers, each data point in output features is computed from multiple points in input features. Hence these layers often have high reuse frequency. Type-I layers are in charge of the main computation of



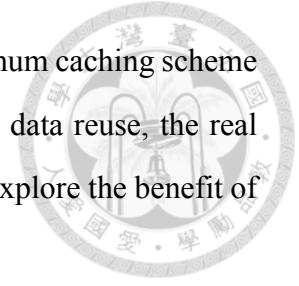
the DNN, so all DNNs must have type-I layers, and they can act as an indicator for the operational intensity of a whole workload. The *type-II layers*, which are on the left of the dashed line in Fig. 5.1, are point-wise layers like batch normalization, activation, add, etc. They are in charge of light-weighted operations. Each data point in output features is computed from single point in input features. Therefore, each data point in input features is only accessed one or two times.

As we can observe in Fig. 5.1, all DNNs have type-II layers, whose reuse frequencies are extremely low. If the tensors of these layers are not treated properly, the data transfer time would be much longer than the computation time they actually need. As we will show in 5.4, such cases would make them really time-consuming. In fact, type-I layers and type-II layers in DNN are usually nearly alternately ordered. A well-known example is a Convolution-BatchNorm-ReLU block in CNN, in which the convolution layer is followed by a batch normalization and ReLU. If the cache can be made large enough to accommodate tensors of more than one layer, we can exploit adjacent-layer reuse or prefetch the tensors of type-II layers while computing type-I layers so that the stall time while computing type-II layers can be shorter, and the compute utilization can be increased.

Observation 2: ResNet and Transformer are tend to compute-bound, while MobileNet and GNMT are tend to memory-bound. Because all DNNs must have type-I layers which contribute to most of the computation, the reuse frequency of these type-I layers can provide a quick view to the compute utilization. Specifically, as Fig. 5.1a and Fig. 5.1d show, most of the type-I layers of ResNet and Transformer are compute-bound. For these layers, as long as the dataflow is designed properly to capture sufficient intra-layer reuse, the compute utilization while computing these layers can be quite high. As a contrast, in Fig. 5.1b and Fig. 5.1c, a considerable portion of type-I layers of MobileNet and GNMT is memory-bound, so the compute utilization of these workloads would be harder to get high.

As a conclusion to this section, we define *reuse frequency* (Equation 3.3 and Equation 3.4) to present the properties of workloads without hardware factors in this subsection. We use modified roofline model to preliminarily show the balance of computation and

data transfer of workloads. These definitions correspond to the minimum caching scheme mentioned at the end of 4.3.3 applied. With larger cache and more data reuse, the real roofline model would be more optimistic than Fig. 5.1, and we will explore the benefit of large cache in the following sections.



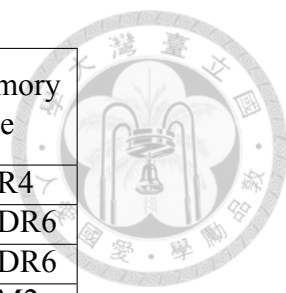
5.3 Overall Performance

As we showed the characteristic of networks and the reuse frequency of layers in 5.2, here we further take the cache capacity into consideration to observe the effect of cache. Table 5.2 lists the configuration of four existing hardware, where Intel i9-10980XE is a CPU, and the others are GPUs. Because our framework aims at modeling large cache capacity, our mechanism has a lower bound for the cache capacity and cannot be arbitrarily low. Hence, we start the cache capacity from 24 MB, which is approximately the same as i9-10980XE last-level cache (LLC), with 2 MB per step up to 1000 MB. And the throughput and bandwidth are mainly taken from the ones of RTX 2080 Ti [25]. By increasing the cache capacity, we would observe how would the performance changes as cache increases in this subsection.

The first thing that the cache capacity can affect is the amount of data reuse, and the performance metric that the most closely related to it is the *memory traffic*. Therefore, we would first show the memory traffic of these workloads with increasing cache capacity in 5.3.1. The second thing that the cache capacity affect is the prefetching distance, the distance that the tensor can be prefetched before it is actually used, which further affects the *execution time*. We would show the execution time with increasing capacity in 5.3.2. Last, as we introduced in 4.2.5 that we use memory traffic and execution time to estimate the *average bandwidth*, thus we put the result of average bandwidth in 5.3.3.

5.3.1 Memory Traffic

Intuitively, a larger cache can provide more opportunity of data reuse, and results in less memory traffic. Fig. 5.2 depicts the trend of memory traffic starting from 24 MB to



| Hardware | Peak FP32 FLOPs | LLC size (MB) | off-chip mem BW (GB/s) | Memory Type |
|----------------|-----------------|---------------|------------------------|-------------|
| i9-10980XE | 2.765 T | 24.75 | 94 | DDR4 |
| RTX 2080 Ti | 13.45 T | 5.5 | 616 | GDDR6 |
| Radeon 6900 XT | 23.04 T | 128 | 512 | GDDR6 |
| A100 | 19.45 T | 40 | 1555 | HBM2e |

Table 5.2: Existing hardware configuration

1000 MB. We plot this figure to point out the effect of increased data reuse by increasing cache capacity. The "In" in Fig. 5.2 represents the memory traffic from off-chip memory into the cache, and the "Out" represents the traffic from the cache out to the off-chip memory. If a tensor already has a copy in the off-chip memory, it can be directly removed without offloading. Hence, the difference between in-traffic and out-traffic can imply the number of repeated re-fetch of tensors. For example, the weights in GNMT repeatedly loaded T times, would result in a difference of $(T - 1) \times cap(W)$.

Observation 3: Adjacent-layer and block reuse can save around 70% of the memory traffic in feature-dominant networks. We start this observation from Transformer, which has the simplest network structure among the four workloads. In Fig. 5.2d, compared with the memory traffic at 24 MB, the memory traffic drops by 67% when cache capacity is 128 MB, which is the capacity of Radeon 6900 XT last-level cache (LLC) [4]. Furthermore, With equations for estimating capacity requirement mentioned in 3.3.2 and 3.3.3, we found that almost all the adjacent-layer reuse and block scale reuse can be captured when the cache is up to 180 MB.

Then we move forward to the CNNs. Because image-based applications often apply down-sampling to identify features of different scales, capacity of tensors in image classification networks varies a lot along the network, ranging from 6 MB to 98 MB in ResNet-50, whereas feature maps in Transformer and GNMT almost keep at the same size. Therefore, as Fig. 5.2a and Fig. 5.2b show, ResNet and MobileNet require more capacity to capture all the adjacent-layer reuse and block scale reuse because of the larger capacity of feature maps in early layers. But one thing in common is that as long as the cache capacity becomes large enough (around 296 MB for ResNet, and 442 MB for Mo-

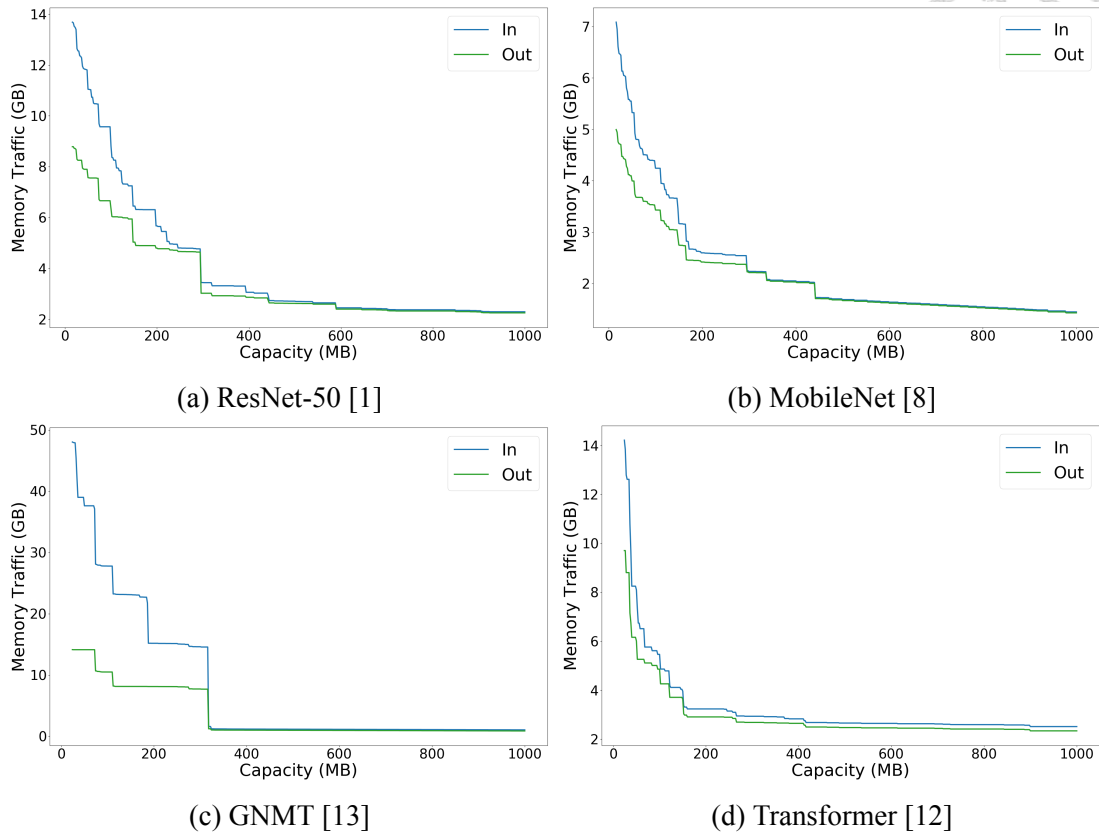
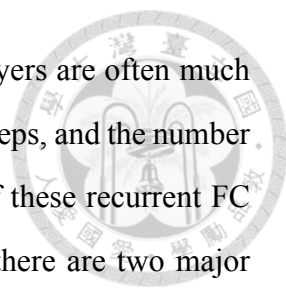


Figure 5.2: Memory traffic in a training iteration with cache capacity from 24 MB to 1000 MB.

MobileNet), the memory traffic drops by around 70% comparing with the one at 24 MB. The saved memory traffic can be even more for inference. Without back propagation, [21] claims that 95% of memory traffic can be reduced in their work, which runs VGGNet inference. However, the method [21] use to exploit adjacent-layer reuse can only work for linear DNNs. Most DNNs proposed later than ResNet [1] include residual connection, and cannot apply fused-layer [21].

However, the mechanism in GNMT is totally different from the others, which we will introduce in the next observation.

Observation 4: Among all types of reuse, recurrent weight reuse is the dominant factor to memory traffic in RNN-based models. As we further investigate into GNMT, we found that the memory traffic at low cache capacity is mainly caused by the FC weights of RNN cells. Recall that we showed in Fig. 3.4, the computation of RNN is step by step feeding the feature maps of different time step to the RNN cell, and the RNN cell, which contains several layers inside, repeatedly uses the same weight while computing different time



steps. Because of such property, the feature maps fed into the FC layers are often much smaller than the weight because they contain only one out of T time steps, and the number of operations is also low. This results in the low reuse frequency of these recurrent FC layers, which can be observed in Fig. 5.1c. To be more specific, there are two major differences between RNNs and other type of networks. First, *RNNs are weight-dominant* in terms of memory traffic because the capacity of weight is larger than that of the feature maps of a time step.¹ Second, *weight will be used multiple times in an inter-layer manner* because of its recurrent structure.

Conclusively, whether weight can be kept in cache to be reused remains a key factor to the performance of RNN-based model. If the cache is not large enough, the weights have to be repeatedly re-fetched, incurring huge inbound traffic. As Fig. 5.2c shows, the memory traffic decreases by 96.6% as cache capacity increased from 24 MB to 500 MB, which demonstrates the influence of the recurrent weight reuse in RNN.

5.3.2 Execution Time

In Fig. 5.3, we tested two different setting of throughput, and the speedup is also plotted. The two lines share exactly the same hardware parameters except the throughput. By comparing the execution time with two different computational throughput, we can observe the extent that the networks are tend to be compute-bound or memory-bound. Memory-bound networks would have limited improvement with merely increasing the computation throughput. On the other hand, compute-bound networks would have their speedup closer to the theoretical bound.

Observation 5: Even the computation-intensive networks cannot fully utilize the increased computation throughput at existing cache capacity. As Fig. 5.3 shows, with 73% more throughput, there is only 20% speedup on ResNet-50 and 40% speedup on Transformer with 24 MB cache. If we look back to Fig. 5.1, all workloads contain some layers whose reuse frequencies are far less than the ridge point, e.g., BatchNorm and Activation. These layers require unattainably high bandwidth (up to 10^5 GB/s) to balance

¹In terms of memory footprint, the total capacity of feature maps is still larger than the weights

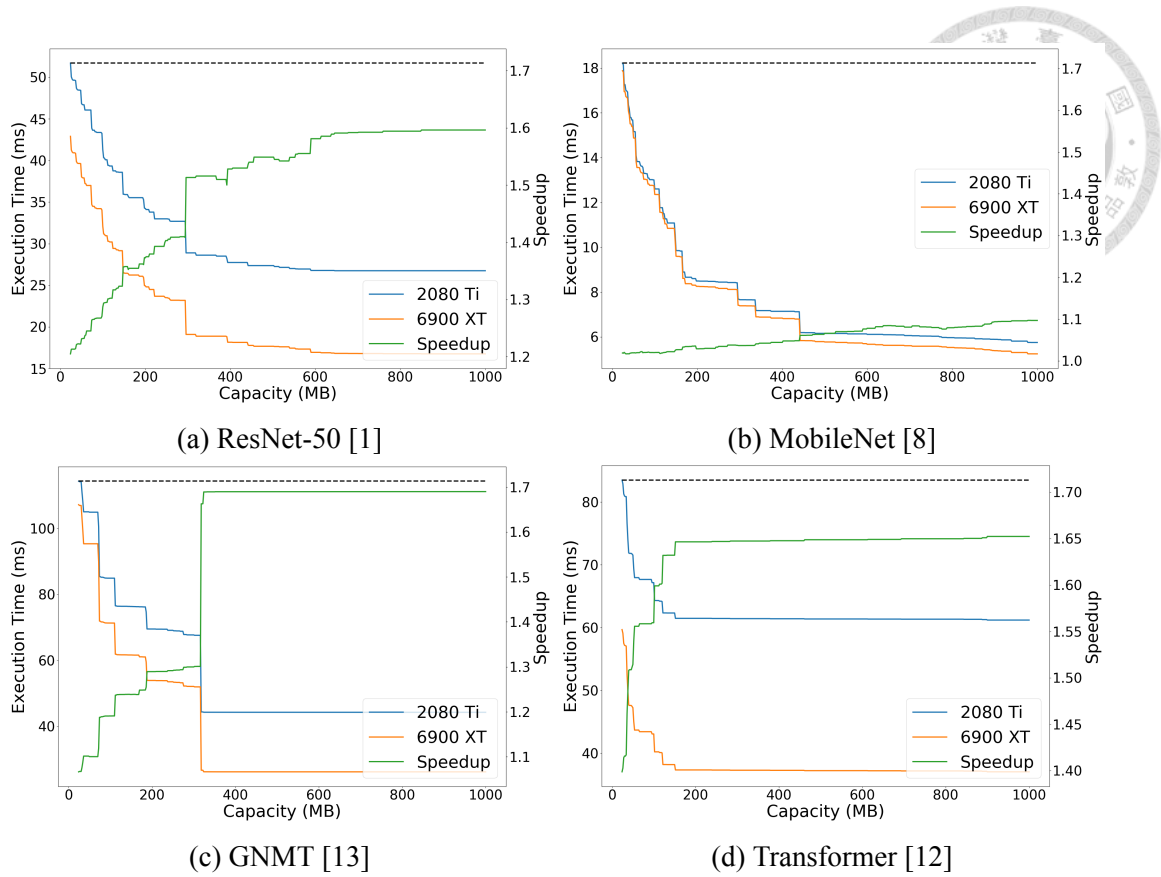
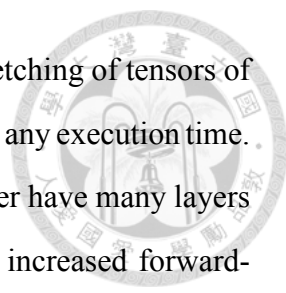


Figure 5.3: Execution time (left axis) of a training iteration with cache capacity from 24 MB to 1000 MB. The throughput of 2080 Ti is 13.45 TFLOPs, and 6900 XT is 23.04 TFLOPs. The speedup (right axis) is the execution time of 2080 Ti divided by the one of 6900 XT. The dashed line at the top is the theoretical upper bound of the speedup, which is $\frac{23.04}{13.45} = 1.71$.

computation and data transfer time, and there is no way to hide the data transfer latency of these layers if the cache is too small to cache a whole layer to exploit adjacent-layer reuse.

Observation 6: Forward-backward reuse helps little for most DNNs. As Fig. 5.3 shows, the lines of all networks except MobileNet almost keep flat after around 400 MB, and the speedup is close to the upper bound at high capacity. When we use the formulas in 3.3 to estimate the capacity at each level of reuse, 400 MB is at the level of block scale reuse, and forward-backward reuse requires GBs of cache to fully capture. This implies that the compute utilization has become almost full without totally having forward-backward reuse.

To explain this phenomenon, as we mentioned in 3.4, data reuse and prefetching are two of the methods that can improve the utilization at low reuse frequency layers. If



the type-I layers have sufficient computation time to overlap the prefetching of tensors of type-II layers, then the increased forward-backward reuse will not save any execution time. For example, as Fig. 5.1a and Fig. 5.1d show, ResNet and Transformer have many layers whose reuse frequency is much larger than the ridge point. So the increased forward-backward reuse at high capacity barely decrease the execution time. Although in Fig. 5.1c it seems that GNMT does not have as many compute-bound layers as ResNet and Transformer, the figure cannot present the effect of recurrent weight reuse. With recurrent weight reuse, in fact most of the feature maps of GNMT are compute-bound. However, for light-weighted workload like MobileNet, this observation does not work well. We will introduce more details in the next observation.

Observation 7: Forward-backward reuse still helps the light-weighted DNNs. In Fig. 5.3, a major difference between MobileNet and the other networks is that the execution time is still decreasing in visible pace at high capacity. As we showed in Fig. 5.1b, most of the Conv2D and DepthwiseConv2D layers in MobileNet have their reuse frequency lower than the ridge point, so the idea that overlapping prefetching with computation-intensive layers mentioned in Observation 6 does not work for MobileNet. In such cases of light-weighted networks, any scopes of reuse mentioned in 3.3 contributes to the execution time because the data transfer time cannot be hidden with the computation.

Observation 8: Memory-bound DNNs are more sensitive to increased capacity. From the characterization shown in Fig. 5.1, we can roughly categorize ResNet and Transformer as compute-bound, and GNMT and MobileNet as memory-bound. Comparing with the 24 MB cache, as the capacity being increased to 500 MB, the execution time of ResNet-50 decreases by 48.1%, and Transformer 26.5%. On the other hand, the time of MobileNet decreases by 69.6%, and GNMT 81.1%. This points out that the benefit that increasing capacity can bring is still closely related to the workload property.

Observation 9: Cache must be large enough to be effective for RNN-based networks. As Fig. 5.3c shows, when the cache is only 24 MB, the speedup of GNMT is just as low as MobileNet, less than 10% out of 71% more computational throughput. Moreover, [19] even reported that the impact of on-chip cache for RNNs is negligible. However, as we

can observe in Fig. 5.1c, the weights of FC in RNN cell without recurrent weight reuse are memory-bound. So the main reason that RNN cannot perform well on state-of-the-art cache is the poor reuse of weights in RNN cells. If we have a large cache, or having a dedicated weight cache to make an attempt to reuse the weights, the performance of RNN can probably be improved.

5.3.3 Average Bandwidth

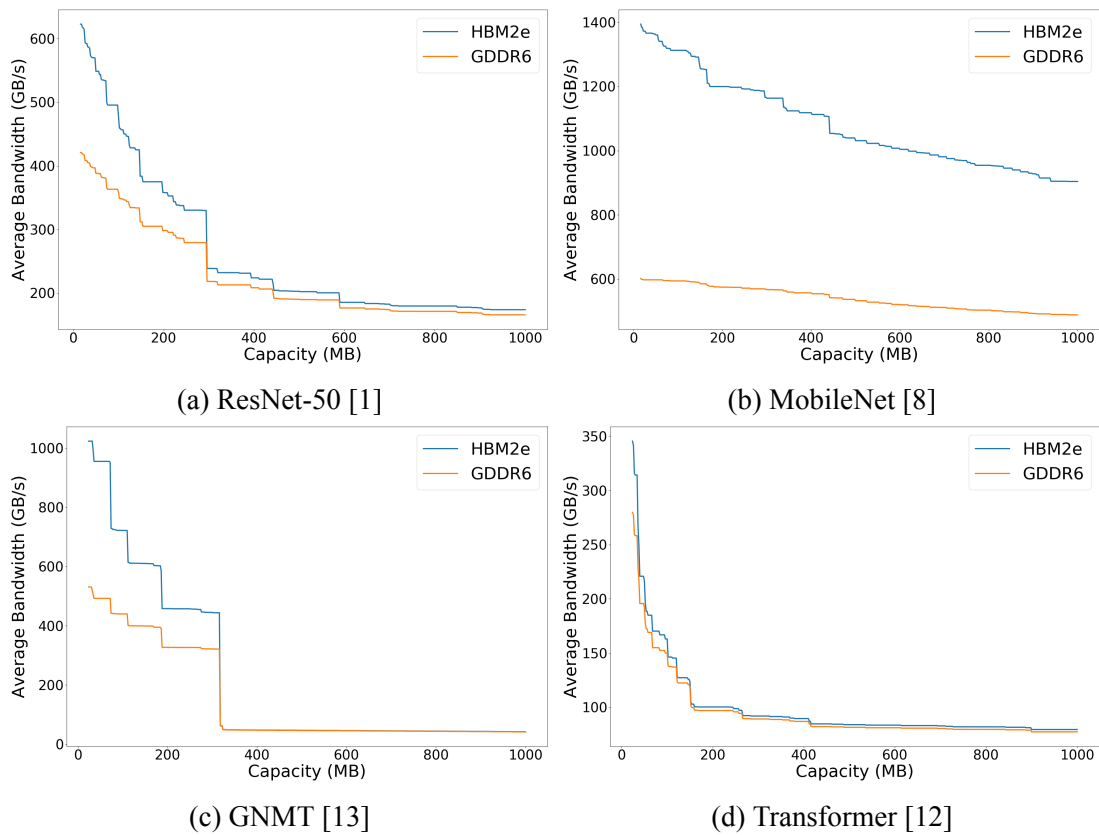


Figure 5.4: Average bandwidth of a training iteration with cache capacity from 24 MB to 1000 MB. The throughput of both lines are set to be 13.45 TFLOPs, which is the same as RTX 2080 Ti. The maximum bandwidth of GDDR6 is 616 GB/s, and that of HBM2e is 1555 GB/s

As Equation 4.1 shows, the average bandwidth is related to total bytes of data transfer and execution time. That is, the results in this part are basically derived from 5.3.1 and 5.3.2. Fig. 5.4 shows the average bandwidth throughout an iteration with different cache capacity. The two lines represents different off-chip bandwidth settings. One is set to be the same as GDDR6 in RTX 2080 Ti, 616 GB/s, and the other is as HBM2e in A100, 1536

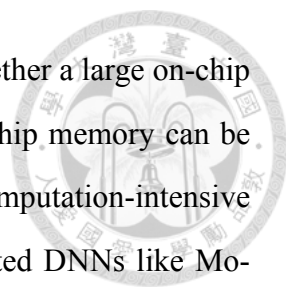
GB/s. Because the memory traffic is not affected by off-chip bandwidth, and the average bandwidth is derived from memory traffic and execution time, the distance between the two lines can also reflect the improvement of execution time that high-bandwidth memory can bring.

Observation 10: For DNNs having enough compute-bound layers, a large cache can diminish the bandwidth requirement. Fig. 5.4 shows that except for MobileNet, the two lines are getting close as capacity increases, which means that the help of HBM becomes smaller. In fact, this observation can be seen as an extension from Observation 6. As long as the prefetching can be overlapped with computation, the bandwidth requirement is effectively reduced.

In conclusion, for these workloads, if the cache is made larger, the off-chip bandwidth does not necessarily have to be high. Bandwidth and capacity are often tradeoff in memory system design. Without the requirement of high off-chip bandwidth, using a low bandwidth but high capacity off-chip memory can become an alternative. The increased off-chip memory capacity can allow deeper DNNs to be trained.

Observation 11: Increasing capacity helps little to alleviate the bandwidth eagerness of light-weighted DNNs. MobileNet is the only one in Fig. 5.4 whose two lines are always far apart even with large cache. Despite the fact that the memory traffic decreases by around 70% with adjacent-layer reuse and block scale reuse in all workloads as we mentioned in Observation 3; however, the execution time of MobileNet also decreases by around 70%, so the average bandwidth of the memory is almost equally high.

To explain this phenomenon, as Fig. 5.1b shows, there are only a few layers whose reuse frequencies are larger than the ridge point. Therefore, in such light-weighted networks, the data transfer time is longer than the computation time in most layers. Even though the cache capacity is increased, possibly allowing longer prefetching distance, there is still not enough computation time to overlap with the prefetching time. These evidences are all telling that this setting of computational throughput is just too high for MobileNet. This workload prefers low computational throughput while high bandwidth configurations.



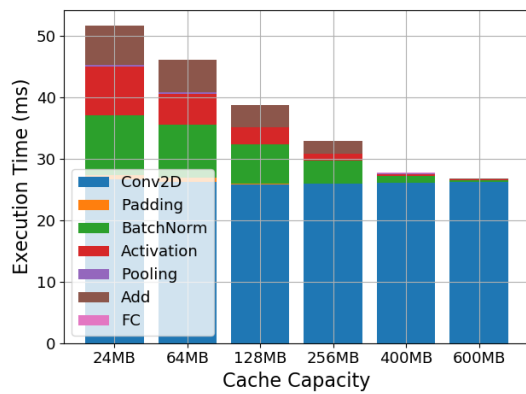
As a conclusion to this section, one of our goals is to explore whether a large on-chip cache can compensate the low off-chip bandwidth, so that the off-chip memory can be easily made larger or cheaper. The answer would be a "yes" to the computation-intensive DNNs like ResNet and Transformer, and a "no" to the light-weighted DNNs like MobileNet. And for RNN-based network, the cache must be large enough to be effective.

5.4 Layer Execution Time Breakdown

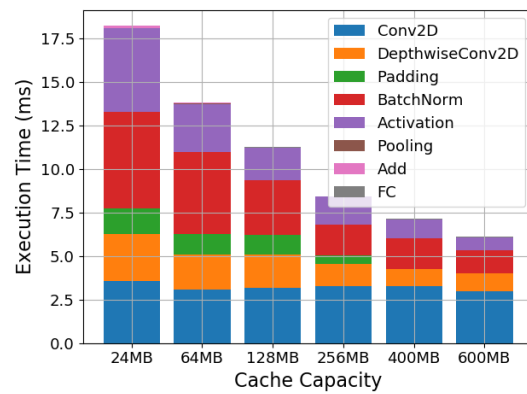
To further analyze the influence of each type of layers, in Fig. 5.5 we extend the results from Fig. 5.3, and breakdown the execution time by the layer type. According to the Table 5.2, we select 24 MB and 128 MB to observe the existing small cache and large cache. And from Fig. 5.3, the lines of most networks are getting flat after 300-400 MB, so we set another point to observe at 400 MB. The other points in between are inserted to observe the trend.

Observation 12: The point-wise layers still take non-negligible part of time in existing cache capacity. When cache capacity is 24 MB, type II layers in ResNet-50 (BatchNorm, ReLU, Add) takes 46.15% of execution time, and those in Transformer (BatchNorm, Activation, Dropout, Add) takes 26.69%. Besides our experiments, [16] also reports that batchnorm and activation takes 26.05% of total execution time when training ResNet-50 on NVIDIA Quadro P4000 GPU with MXNet. As we introduced in 4.2.5, execution time is related to both computation and data transfer. Although point-wise layers are having very low computation time, the data transfer time can also make them time-wasting. As the cache capacity increases, the execution time taken by these point-wise layers can be effectively reduced by reuse and prefetching.

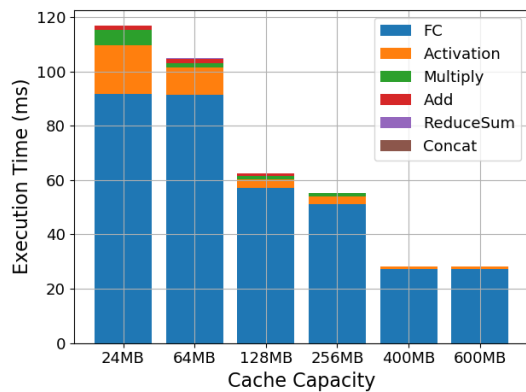
Observation 13: The data transfer time of type-II layers can be almost perfectly hidden when the cache capacity is at the level of caching a block of layers. As Fig. 5.5a and Fig. 5.5d show, as the capacity increases, the reduced execution time in ResNet and Transformer mainly results from the type-II layers. Type-I layers have longer computation time than data transfer time, so they are barely affected by the increased capacity. The



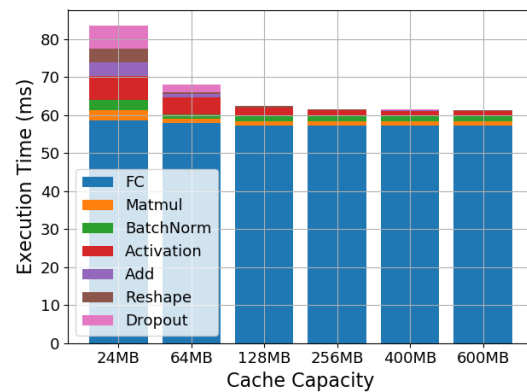
(a) ResNet-50



(b) MobileNet



(c) GNMT



(d) Transformer

Figure 5.5: Execution time breakdown with different cache capacity

execution time is dominated by type-I layers in both networks when the capacity is 400 MB, which also provides an evidence to Observation 6 that increasing data reuse at this level would not improve the execution time of compute-bound layers.

5.5 Effect of Batch Size

Batch training is a widely used skill in DNN training. Let alone the accuracy, batch training is in fact a double-edged sword. With larger batch size, the reuse frequency of weight can be increased, but the capacity of feature map also increases, enlarging the capacity requirement of the same reuse distance. Fig. 5.6 shows the effect of batch size under low and high cache capacity. The metric to compare performance among different batch size, throughput (sample/s), is calculated by Equation 5.2, which means the the

number of samples that can be trained by the network per second.

$$\text{Throughput} = \frac{\text{batch size}}{\text{execution time of an iteration}} \quad (5.2)$$

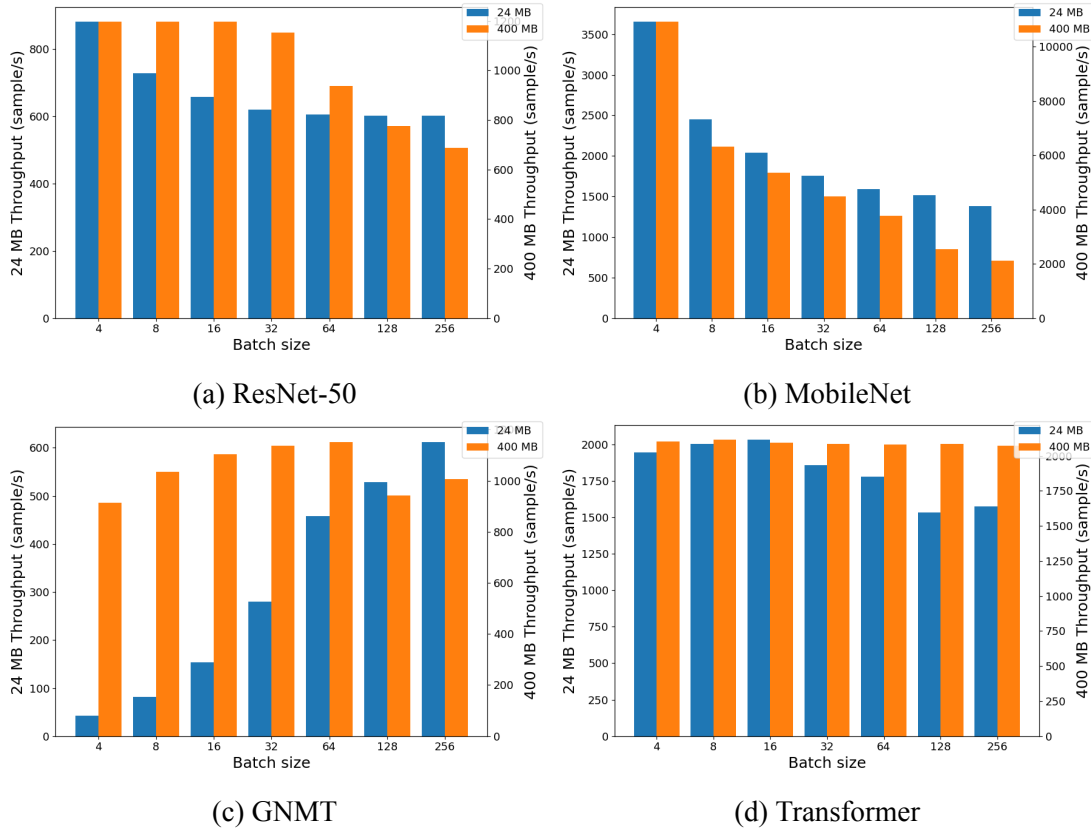
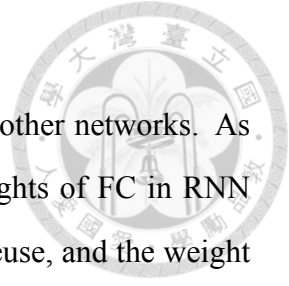


Figure 5.6: Throughput at different batch size. The blue bars (left axis) represent the throughput when cache is 24 MB, and the orange bars (right axis) represent the throughput when cache is 400 MB.

Observation 14: Increasing batch size does not always increases the throughput. As Fig. 5.6 shows, for ResNet, MobileNet, and Transformer, the increased reuse frequency brought by increasing batch size is negligible. The limiting factor for these networks are the data reuse that can be captured. As we introduced in Observation 6, whether ResNet and Transformer can have high utilization depends on whether adjacent-layer reuse and block scale reuse can be captured. If the capacity of feature maps is too large for these reuse, the latency at point-wise layers cannot be hidden. On the other hand, for MobileNet, as we mentioned in Observation 7, the capacity of feature map affects the proportion of forward-backward reuse that can be captured. So MobileNet prefers the batch size to be

as small as possible.

However, the mechanism in GNMT is totally different from the other networks. As we mentioned in Observation 9, unlike the other networks, the weights of FC in RNN cells do not have enough reuse frequency without recurrent weight reuse, and the weight is the major limiting factor. Therefore, when the cache is small, increasing batch size can increase the reuse frequency of these weights thus improves the throughput. With 400 MB cache, if the batch size is too large so that some of the recurrent weight reuse is missed, the throughput will be lower than with small batch size.





Chapter 6

Conclusion

We introduce all scale of data reuse throughout a DNN training iteration, and we presented an analytical performance modeling tool for DNN training. We use this tool to explore the data reuse can be captured at a given cache capacity, and the maximum overlapping of computation and data transfer given computational throughput and off-chip bandwidth. With the help of this tool, we can explore the benefit of a large cache can bring, or discuss what may happen with a different setting of off-chip bandwidth or computational throughput, which can allow us to discuss a suitable logic/memory balance, or even use it for design space exploration (DSE).

From our experiment results, we found that cache indeed plays an important role in DNN training, though the benefit it brings is closely related to the workload characteristics. For computation-intensive workloads, the bottleneck is whether the latencies of the point-wise layers can be hidden through adjacent-layer reuse or prefetching. For light-weighted workloads, the key problem is to select an appropriate combination of computational throughput and bandwidth that fits the reuse frequency of computation layers. For RNN-based models, the cache must be large enough to cache recurrent weight to be helpful. And when it comes to batch size, one must consider both the increased reuse of weights and increased capacity of feature maps.

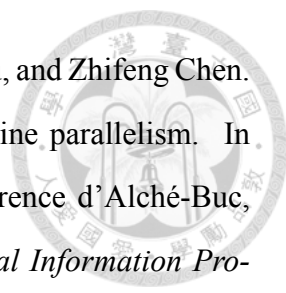
From the view of progress of IC process, having a large cache should be definitely possible, and more and more design options are appearing with emerging memories. For example, a large cache can decrease data transfer requirements, and allow off-chip mem-

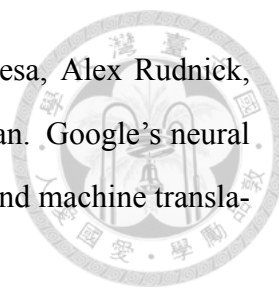
ory to be made with denser but lower bandwidth memories, which can allow deeper DNN or larger batch size to be trained. Thus we explore the performance with a larger cache with representative workloads to find out the effect of cache capacity to data reuse and I/O latency hiding. Moreover, this tool allows accelerator hardware designers to evaluate tradeoffs between logic and memory. It also provides network researchers to quickly get some feedback on the efficiency of new models on accelerators.



Bibliography

- [1] Kaiming He, X. Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.
- [2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1, NIPS'12*, page 1097–1105, Red Hook, NY, USA, 2012. Curran Associates Inc.
- [3] J. Chang, Y. Chen, G. Chan, H. Cheng, P. Wang, Y. Lin, H. Fujiwara, R. Lee, H. Liao, P. Wang, G. Yeap, and Q. Li. 15.1 a 5nm 135mb sram in euv and high-mobility-channel finfet technology with metal coupling and charge-sharing write-assist circuitry schemes for high-density and low-vmin applications. In *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, pages 238–240, 2020. doi: 10.1109/ISSCC19947.2020.9062967.
- [4] Amd radeon rx 6900 xt. URL <https://www.amd.com/en/products/graphics/amd-radeon-rx-6900-xt>.
- [5] Nvidia rtx 3090, . URL <https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-3090/>.
- [6] Nvidia rtx 3090 specs, . URL <https://www.techpowerup.com/gpu-specs/geforce-rtx-3090.c3622>.
- [7] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu

- 
- Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 103–112, 2019. URL <https://proceedings.neurips.cc/paper/2019/hash/093f65e080a295f8076b1c5722a46aa2-Abstract.html>.
- [8] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018.
- [9] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection, 2020.
- [10] keras.applications. URL <https://keras.io/api/applications/>.
- [11] Joseph Redmon. Darknet: Open source neural networks in c. <http://pjreddie.com/darknet/>, 2013–2016.
- [12] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30, pages 5998–6008. Curran Associates, Inc., 2017. URL <https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf>.
- [13] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian,



Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation, 2016.

- [14] H. Kwon, P. Chatarasi, V. Sarkar, T. Krishna, M. Pellauer, and A. Parashar. Maestro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings. *IEEE Micro*, 40(3):20–29, 2020.
- [15] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers. Accel-sim: An extensible simulation framework for validated gpu modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 473–486, 2020.
- [16] H. Zhu, M. Akrouf, B. Zheng, A. Pelegris, A. Jayarajan, A. Phanishayee, B. Schroeder, and G. Pekhimenko. Benchmarking and analyzing deep neural network training. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 88–100, 2018. doi: 10.1109/IISWC.2018.8573476.
- [17] Ananda Samajdar, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. Scale-sim: Systolic cnn accelerator simulator. *arXiv preprint arXiv:1811.02883*, 2018.
- [18] Baidu deepbench. URL <https://github.com/baidu-research/DeepBench>.
- [19] Aajna Karki, Chethan Palangotu Keshava, Spoorthi Mysore Shivakumar, Joshua Skow, Goutam Madhukeshwar Hegde, and Hyeran Jeon. Tango: A deep neural network benchmark suite for various accelerators, 2019.
- [20] K. Siu, D. M. Stuart, M. Mahmoud, and A. Moshovos. Memory requirements for convolutional neural network hardware accelerators. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 111–121, 2018. doi: 10.1109/IISWC.2018.8573527.
- [21] M. Alwani, H. Chen, M. Ferdman, and P. Milder. Fused-layer cnn accelerators. In

2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 1–12, 2016. doi: 10.1109/MICRO.2016.7783725.



- [22] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009. ISSN 0001-0782. doi: 10.1145/1498765.1498785. URL <https://doi.org/10.1145/1498765.1498785>.
- [23] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: Dynamic gpu memory management for training deep neural networks. *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb 2018. doi: 10.1145/3178487.3178491. URL <http://dx.doi.org/10.1145/3178487.3178491>.
- [24] X. Chen, D. Z. Chen, Y. Han, and X. S. Hu. modnn: Memory optimal deep neural network training on graphics processing units. *IEEE Transactions on Parallel and Distributed Systems*, 30(3):646–661, 2019. doi: 10.1109/TPDS.2018.2866582.
- [25] Nvidia rtx 2080 ti specs, . URL <https://www.techpowerup.com/gpu-specs/geforce-rtx-2080-ti.c3305>.
- [26] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debojyoti Dutta, Udit Gupta, Kim Hazelwood, Andrew Hock, Xinyuan Huang, Atsushi Ike, Bill Jia, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Guokai Ma, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Tsuguchika Tabaru, Carole-Jean Wu, Lingjie Xu, Masafumi Yamazaki, Cliff Young, and Matei Zaharia. Mlperf training benchmark, 2020.
- [27] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao,

T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 446–459, 2020. doi: 10.1109/ISCA45697.2020.00045.