# Efficient in-hardware compression of on-chip data

by

Amin Ghasemazar

M.Sc., University of Tehran, 2015
B.Sc., Iran University of Science and Technology, 2012

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

December 2021

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the dissertation entitled:

**Efficient in-hardware compression of on-chip data**

submitted by **Amin Ghasemazar** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**
in **Electrical and Computer Engineering**

**Examining Committee:**

Mieszko Lis, Electrical and Computer Engineering, UBC

---

Co-supervisor

Prashant Nair, Electrical and Computer Engineering, UBC

---

Co-supervisor

Alexandra Fedorova, Electrical and Computer Engineering, UBC

---

Supervisory Committee Member

Shahriar Mirabbasi, Electrical and Computer Engineering, UBC

---

University Examiner

Frank Wood, Computer Science, UBC

---

University Examiner

Alaa Alameldeen, School of Computing Science, SFU

---

External Examiner

# Abstract

The past decade has seen tremendous growth in how much data is collected and stored, and consequently in the sizes of application working sets. On-chip memory capacities, however, have not kept up: average CPU last-level cache capacity per core (thread) has stagnated at 1MB. Similar trends exist in special-purpose computing systems, with only up to tens of megabytes of on-chip memory available in most recent AI accelerators.

In this dissertation, we explore hardware-friendly online data compression techniques to gain the performance benefits of larger on-chip memories without paying the costs of larger silicon. We propose several solutions including two methods to compress general workloads in CPU caches, and two methods to compress AI workloads in special-purpose computing systems.

In order to efficiently compress on-chip data, the compression mechanisms need to leverage all relevant data stored in on-chip memory. We propose 2DCC, a cache compression mechanism that leverages redundancy both within and across all cache data blocks, and results in a 2.12× compression factor. We then extend this insight by observing that many on-chip blocks are often similar to each other. We propose Thesaurus, a hardware-level on-line cacheline clustering mechanism to dynamically form clusters as these similar blocks appear in the data access stream. Thesaurus significantly improves the state-of-the-art cache compression ratio to 2.25×.

Next, we apply our insights to special-purpose applications. We first propose Channeleon, which tackles the problem of compressing the activation maps in deep neural networks (DNNs) at inference time. Leveraging the observed similarity among activation channels, Channeleon first forms clusters of similar activation channels, and then quantizes activations within each cluster. This enables the activations to have low bit-width while incurring acceptable accuracy losses. Lastly, we propose Procrustes, a sparse DNN training accelerator that prunes weights by exploiting both software and hardware knowledge. Procrustes reduces the memory footprint of models by an order of magnitude while maintaining dense model accuracy.

# Lay Summary

This dissertation explores the potential to improve the performance and efficiency of workloads through efficient representation of data. It proposes workload acceleration for both general-purpose workloads and machine learning workloads. We first address missed opportunities in the general-purpose systems and propose techniques for in-hardware compression of these workloads which makes them run faster compared with when using the existing methods. Then, we propose two compression techniques for machine learning workload memory footprint reduction and acceleration.

# Preface

This section lists my publications and posters that were completed at The University of British Columbia in chronological order and form a part of this thesis work. This section finishes by highlighting my contributions to this dissertation.

The publications and posters that are incorporated in this thesis are listed as follows:

**[C1]** A. Ghasemazar, M.Ewais, P.Nair, M. Lis,"Cache Compression in Two Dimensions," IEEE 37th International Conference on Computer Design (ICCD) , 2019.

**[C2]** A. Ghasemazar, M. Ewais, P. Nair, M. Lis, "2DCC: Cache Compression in Two Dimensions," In Design, Automation and Test in Europe Conference and Exhibition (DATE), 2020.

**[C3]** A. Ghasemazar, M. Ewais, M. Lis, "Decoupling Approximation and Cache Compression," The 2020 Workshop on Approximate Computing Across the Stack (WAX), ASPLOS workshop, 2020.

**[C4]** A. Ghasemazar, P. Nair, M. Lis, "Thesaurus: Efficient Cache Compression via Dynamic Clustering," In The International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2020.

**[C5]** D. Yang, A. Ghasemazar, X. Ren, M. Golub, G. Lemieux, M. Lis, "Procrustes: a Dataflow and Accelerator for Sparse Deep Neural Network Training," In The International Symposium on Microarchitecture (MICRO), 2020.

## Thesis Outline

Chapter 2. This chapter combines and expands on the material presented in [C1], [C2], [C3], [C4], and [C5] to describe the necessary background information for this dissertation.

Chapter 3. A version of this material has been published as [C1], and presented as [C2] and [C3]. In [C1], [C2], and [C3], I was the lead re-

searcher responsible for conducting the majority of the research and writing the manuscript under the guidance of Dr.Mieszko Lis and Dr.Prashant Nair. Mohammad Ewais was responsible for the functional integration of the prior work into the simulator, as well as an initial implementation of the proposed method. I was responsible for the detailed timing implementation, integration, and evaluation of the prior work as well as optimizing the proposed method implementation for better performance, conducting most of the experiments, analyzing the results, and writing the corresponding portions of the manuscript.

Chapter 4. A version of this material has been published as [C4]. In [C4], I was the lead researcher responsible for conducting the research, implementing the proposed method in the simulator, performing the experiments, collecting and analyzing the results, and writing the manuscript under the guidance of Dr.Mieszko Lis and Dr.Prashant Nair.

Chapter 5. I was the lead researcher for this work and responsible for the background research, the majority of the algorithm and design, most of the writing, and parts of the implementation. Dingqing Yang, was responsible for the majority of the implementation and was also involved with the methodology.

Chapter 6. A version of this material has been published as [C5]. in [C5], I was the second author responsible majorly for the software and algorithm side of the work. Specifically, I was responsible for some background research, the implementation of the machine learning models including reproducing the prior work on sparse training, extending the software framework to support the proposed mechanisms, understanding the trade-offs for hyperparameter search and tuning them to work with the proposed methods, conducting the experiments to collect and analyze model performance and accuracies, and writing the corresponding sections in the manuscript. I was also partly involved with the design of the hardware architecture of the proposed work. The lead author, Dingqing Yang, and the other author, Xiaowei Ren, were responsible for the majority of the background research, the dataflow and data layout choices and trade-offs in the hardware, the sparse data compression format, algorithm adaption to make it hardware-friendly, most of the architectural decisions, all the hardware implementation, conducting hardware simulation experiments, and writing the manuscript for the corresponding sections.

Chapter 7. This chapter combines and expands on the related work sections that were presented in [C1], [C2], [C3], [C4], and [C5] with the related work for the work in Chapter 6.

# Table of Contents

# List of Tables

# List of Figures

# Abbreviations

**ALU** Arithmetic logic unit

**B$\Delta$I** Base-delta immediate

**CNN** Convolutions neural network

**CONV** Convolutional

**CPU** Central processing unit

**CSB** Compressed sparse block

**CSC** Compressed sparse column

**DNN** Deep neural network

**DRAM** Dynamic random-access memory

**FC** Fully connected

**GLB** Global buffer

**IPC** Instructions per cycle

**LLC** Last-level cache

**LRU** Least recently used

**LSH** Locality-sensitive hashing

**MAC** Multiply and accumulate

**MPKI** Misses per 1000 instructions

**MSE** Mean squared error

**PE** Processing element

**PTQ** Post-training quantization

**QoR** Quality of result

**ReLU** Rectified linear unit

**RF** Register file

**RNG** Random number generator

**SRAM** Static random-access memory

# Acknowledgements

None of this work would have been possible without the support of my family, advisors, colleague, and peers. First, I would like to thank my advisors Mieszko Lis, and Prashant Nair for many years of guidance, patience, motivation, and support. Thank you for providing an environment for me to fail and grow and an opportunity to pursue my ideas. Thank you for your invaluable insights into our work, research, and all the days and nights that you stayed and worked with me. Thank you for always being there whenever I needed you. Thank you for always expecting the best and pushing me to do high-quality research. Your dedication will always inspire me both in my personal and professional life.

I would especially like to thank my family for helping me throughout this journey. To my father – thank you for everything. You inspired me to be the best of me, to become an engineer, and removed many barriers for me to become who I am today. Thank you for your non-stop support. To my mother – thank you for your love and patience. Thank you for all your emotional support, worrying about my worries, and becoming happy on my happy days. I wouldn't be able to overcome ups and downs without your love. To my brothers – thank you for always opening the path for me and being a good example in front of me. Thank you for your dedication in what you do and your achievements, which inspired me to step up my game as well. Thank you for always being someone who I can trust, feel loved, talk to, and feel understood. Thanks to all my relatives, who pumped a lot of positive energies by believing in me and motivating me to pursue my path.

Finally, I thank all my peers and colleagues both in university and outside who helped me overcome challenges during my PhD program. Without their emotional and technical support, this journey would not be enjoyable. To all of the UBC friends I had the pleasure of working with and learning from them – Xiaowei Ren, Jeff Goeders, Dingqing Yang, Mohammad Ewais, Maximilian Golub, Khaled E. Ahmed, Mohamed Omran Matar, John Deppe, Peter Deutsch, Mohammad Olyaiy, Jose Pinilla, Al-Shahna Jamal, Hossein Omidian, Tayler Hicklin Hetherington, Shadi Assadikhomami, Ahmed ElTantawy, Ayub Gubran, Dave Evans – thank you for making this

such a memorable experience. Especially, it was my pleasure to collaborate with Xiaowei Ren, Dingqing Yang, and Mohammad Ewais.

# Dedication

*To my mother, Shamsi, and father, Hassan.*

# Chapter 1

# Introduction

## 1.1 Computing Trends and Storage Requirements

Off-chip memory bandwidth has been considered one of the major limiting factors to processor performance. The "Memory Wall" [257] describes the disparity between the rate of CPU performance improvement and the relatively flat rate of off-chip memory bandwidth increase. Researchers have ameliorated the issue of limited off-chip bandwidth by adding on-chip *caches* in CPUs and on-chip *scratchpads* in HW accelerators. This makes a subset of memory requests, for example, more frequently accessed addresses, accessible at lower latencies. These on-chip memories had been useful for several decades; recently, however, due to big data's exponential growth [162], memory requirements have surpassed on-chip memory capacities. Now, the problem arises from not having enough on-chip memory capacity, which translates to an increased number of accesses to off-chip memories with orders of magnitudes more energy and latency costs [57, 105] compared to on-chip memories.

Limited on-chip memory capacity is a challenging problem to solve. To begin with, it is not practical to increase the chip size to provide more on-chip memory storage space. Increasing the chip size makes the processor unmanufacturable because of the silicon cost and yield problems [145, 244]. Moreover, progress in memory improvement has not kept up with demand for faster and larger memories due to energy and cost limitations [121]. This means that even if we can allocate more silicon, simply increasing the amount of on-chip memory may result in less overall chip efficiency: larger SRAM memories, have larger number transistors, more complex routers, longer wires, and larger sense amplifiers; thus, larger on-chip memories have higher leakage power, higher dynamic power, higher access latency, and higher silicon costs (see Figure 1.1).

Therefore, these challenges necessitate innovative solutions in order to increase on-chip memory capacities without the drawbacks of increasing the

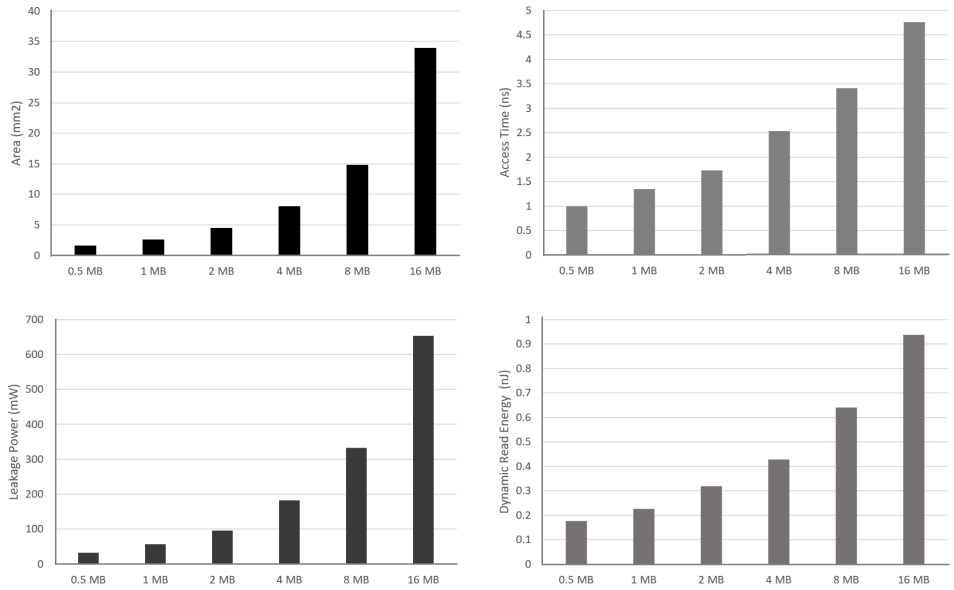



Figure 1.1: Area, leakage power, dynamic power, and access time are affected by the SRAM array size. Measured at $32nm$ technology using CACTI [138].

silicon area.

An immediate alternative could be to use denser memories with the help of emerging memory technologies. Unfortunately, there are some practical limitations in using these memories. Although they can be built in dense and higher capacities, they tend to have high latency and energy consumption, reliability issues, limited bandwidth, and low endurance compared to SRAM [60, 180].

Moreover, currently there is no commercial emerging memory that can be used as a good replacement for current memories. Academic efforts [60, 75, 180, 217, 270] resulted in a few non-production level prototypes [75, 217]; to the best of our knowledge, Intel® Optane™ DC PMM [28] is the only commercial device in production, but, this device suffers from high latency, low bandwidth, and added software complexity in order to use it [130, 224].

Due to these limitations, the only solution left is to efficiently utilize on-chip memory to fit more data on existing storage space. In this thesis, we investigate the problem of increasing effective capacity of on-chip memories through data compression. We use hardware-level efficient data compression methods to store the same amount of information in smaller memory blocks in on-chip memories. Although our techniques are not limited to SRAM and

can be applied to all types of memories including the emerging memories without the loss of generality, here we specifically focus on last-level cache memories in general-purpose computing systems, and scratchpad memories in special-purpose AI accelerators.

State-of-the-art techniques for last-level cache compression can generally fall into two classes: intra-cacheline compression, which places multiple memory blocks within each cacheline [191, 200, 219], and inter-cacheline deduplication [242], which helps capture data redundancy across cacheline boundaries by detecting identical cachelines and storing only a single copy. These methods suffer from either a small compression ratio, high hardware complexity, or large decompression latency.

In the special-purpose computing domain, AI accelerators have attracted special attention in recent years. Deep Neural Networks (DNNs) [152] were developed to identify relationships in high-dimensional data, and now are the driving technology for numerous application domains [65, 66, 99]. Recent DNNs contain several millions [112, 216, 281] or even billions [74, 203] of parameters with gigabytes of memory footprint. The memory footprint can be reduced by leveraging more efficient values (i.e., parameters, activation maps) through various techniques such as pruning and quantization at the cost of consequent accuracy degradation. A major drawback of existing methods is their inability to significantly reduce the memory footprint while preserving the network accuracy.

Although data compression can help tackle the trend of having larger working set sizes, it is not trivial to make data compression work in hardware for these applications.

## 1.2 Challenges

Aside from achieving high compression ratio, the following considerations need to be taken into account when designing a hardware compression method:

- **Low complexity and implementation cost**: To make any compression feasible in hardware, it should require minimal changes to existing software and hardware stack. Significant changes to existing hardware can hinder the integration and implementation of such methods, especially in commercial devices.

  The focus of chapter 3 and chapter 4 is to develop efficient compression methods for existing CPU caches; therefore, we make minimal changes

to the structure of the conventional cache to be able to integrate our compression modules. Furthermore, the compression happens almost entirely in hardware, which means no changes to the software stack are required.

- **Low latency, area, energy, and memory management overheads**: In order for the methods to be more feasible to be implemented in on-chip memories, they should operate at reasonably fast speeds. Many of the applications are latency-critical, and the compression methods should stay within the access times of on-chip memories (only a few cycles). Further, the added area and energy overheads should be low enough so that the benefits of having the compression method surpass the overheads. Having data blocks with various sizes also complicates the space management and block lookup procedures on the memory. Therefore, more advanced replacement policies are needed in order not to waste the memory space and keep the access latency low.

  In chapter 3, we develop a hardware-friendly compression method via a novel combination of two well-known cache compression methods, and develop a novel replacement policy to efficiently manage space in a compressed cache. In chapter 4, we develop a novel in-hardware compression method that captures and compresses similar data blocks by using a dynamic clustering mechanism. We perform the dynamic clustering using a hardware-friendly variant developed based on locality-sensitive hashes (LSH) [127]; it computes the hashes using only addition and subtraction operations. We also propose a cache replacement policy in order to avoid the issue of under-utilization of cache space due to variable-sized blocks. Later in chapter 5, we use sampling before performing the clustering task in order to keep the computation costs low. Lastly, in chapter 6, we replace the sorting algorithm needed for model pruning with an on-the-fly cheap partitioning method. All of these techniques help add compression to on-chip memories with negligible overheads.

- **Zero or low application quality loss**: In most cases applying the compression method to applications should not affect the quality of results (QoR) which is measured by metrics such as accuracy. However, there exist some applications like machine learning applications where relaxing some quality constraints enables higher memory savings. Therefore, application quality is being traded for memory com-

pression, and so the least amount of quality degradation is desired for any given compression ratio.

In chapter 3 and chapter 4, we propose lossless compression methods targeted for general-purpose processors. The decompressed block can be reconstructed to its original value using simple operations such as subtraction and logical XOR. In chapter 5 and chapter 6, however, due to the fact that machine learning models are able to tolerate errors, we use and develop lossy compression methods such as low-bitwidth quantization and pruning to significantly reduce the memory footprint; these methods use domain-specific knowledge such as the statistics of the values, and are specifically designed to keep accuracy losses low.

Considering all these challenges, we describe the insights we leveraged to develop efficient on-chip compression methods in the following section.

## 1.3 Thesis Statement

This dissertation explores the potential to utilize in-hardware compression of on-chip data in order to increase overall system performance without incurring the energy and cost overheads of larger memories. Methods described here enable efficient compression of on-chip data that are dynamic and constantly changing.

First, it is essential to go beyond compressing values in isolation or only a few consecutive data blocks. An effective compression scheme performs the compression across a broader set of blocks by taking advantage of redundancies across and within memory blocks. This essentially enables the compression method to make better compression decisions (i.e., what data blocks can be compressed together, what values can be dropped, etc.) based on the relevant data points in the entire set of available on-chip data.

Second, efficient data compression methods, need to account for not only exact repetitions (duplicates), but also more importantly for approximate repetitions (near-duplicates). We observed that there are similarities among memory blocks stored in on-chip memories both in general-purpose and special-purpose computing systems. This similarity can be used toward developing compression methods that search for these near-duplicate memory blocks throughout the memory, and efficiently stores non identical parts.

Third, dynamic clustering algorithms are well-suited to convert this similarity in on-chip data to storage savings. Similar data blocks within each cluster can be delta encoded with respect to the centroid of each cluster,

therefore, occupying much less memory. Also, as the data comes to on-chip memory when accessed and might be evicted after it is consumed, the clustering and compression should be dynamic at run time.

Finally, efficient in-hardware clustering is crucial to convert storage savings into system performance. The benefits from the compression method should outweigh the cost of having it; complex algorithms that work best (i.e., form better clusters) in software are not necessarily the best choice for in-hardware implementation. On-chip memories are latency and energy-critical, and any compression methods should be hardware-friendly. Clusters can be formed cheaply using approximate similarity search methods, while keeping the compression method lossless for sensitive applications.

## 1.4 Contributions

We propose and develop two novel hardware-level techniques for compressing last-level CPU caches, as well as two techniques for compressing the data in machine learning applications.

In the first work, we identify the missed opportunity in compressing cache memories due to not considering data throughout the on-chip memory when capturing data redundancy. We show that leveraging redundancies within cachelines, or only across cachelines leads to a significant loss in compression opportunities for several applications: some workloads exhibit either inter-block or intra-block redundancy, while others exhibit both. We propose 2DCC, a simple technique that takes advantage of both types of redundancy to compress the data by 2.12× (geomean) over the baseline with no compression. We evaluate 2DCC in an iso-silicon basis compared to the baseline and show that it can cheaply be integrated into existing caches.

Next, we identify a previously untapped source of inefficiency in cache compression methods: not accounting for similar memory blocks that is stored on on-chip memories. we propose Thesaurus, a dynamic inter-cacheline compression technique to efficiently detect and compress groups of memory blocks that have *nearly identical*, rather than exactly identical, data values. The similar data is clustered using a hardware-friendly variant of locality-sensitive hashing. To compress the cache, Thesaurus stores the "clusteroid" of each cluster together with the (much smaller) "diffs" needed to reconstruct the rest of the cluster, which significantly improves cache compression to 2.25× (geomean).

As the third contribution we tackle the problem of data compression in DNNs; more specifically, DNN activation map compression at inference-

time, which can significantly reduce the memory footprint in scratch-pad memories. We observe that many activation channels share similar statistics. Thus, we propose Channeleon, a technique which looks at the entire set of channels at each layer, clusters them based on activation statistics, and performs non-linear quantization on each cluster separately. This method, unlike prior work, does not need to have access to the training data and does not need any tuning. Channeleon is able to compress the activation maps to 5-bit while outperforming the best state-of-the-art method by 60% top-1 accuracy on large-scale visual dataset (ImageNet [71]).

As the final contribution, we propose Procrustes, an AI hardware accelerator that produces compressed sparse weights from scratch based on software-hardware co-design. This accelerator produces an order-of-magnitude compressed models by considering the entire set of gradient values and keeping the ones with the highest change, rather than looking at the gradients or weights individually. In order to make this technique hardware-friendly, we use a computationally simple quantile estimation method to track weights, and also leverage a novel data-flow and load-balancing scheme that converts sparsity into speedups. Procrustes results in up to 3.26× energy savings and up to 4× speedup compared to the state-of-the-art accelerators, while maintaining dense model accuracy.

## 1.5 Organization

The rest of this dissertation is organized as follows:

- Chapter 2 discusses the relevant background information for this dissertation, such as data compression, memory hierarchy in general-purpose and special-purpose systems, as well as compression techniques for on-chip memories.
- Chapter 3 characterizes the workloads and studies the state-of-the-art methods for data compression in cache memories. Then it presents a compression method that leverages redundancy within and across data lines in the cache memory.
- Chapter 4 extends the prior insight on granularity with approximate nearest neighbour search to find similar blocks of data in cache memory. It proposes and evaluates a novel compression technique based on dynamic clustering to compress blocks against similar data lines that are already stored in the cache.

- Chapter 5 identifies the trend in memory usage of modern machine learning models, which is dominated by activation maps, and proposes and evaluates a novel online compression mechanism to reduce the memory footprint of activation maps at inference time.
- Chapter 6 demonstrates co-design approach for accelerating AI training. It adapts a model pruning method that achieves an order of magnitude in weight footprint reduction by considering the entire set of gradients to suit the hardware constraints. Then, it develops dataflow, data layout, load balancing techniques and proposes a sparse training hardware accelerator to convert this compression to speedups and energy gains.
- Chapter 7 discusses the related work for this dissertation.
- Chapter 8 concludes this dissertation and discusses directions for future work.

# Chapter 2

# Background

In this chapter, we give a concise overview of the necessary background topics and materials, and place the rest of the thesis in the content of the existing literature. We begin with an overview of computing systems and their memory hierarchies. Then, we describe the motivation and potential of data compression for on-chip memories. We continue with some background in data compression and outline a few of the traditional compression methods. Next, we discuss the challenges of compression in hardware. Finally, we conclude this section by a brief overview of the state-of-the-art methods and their drawbacks.

## 2.1 Computing Systems

Von Neumann computers, like general-purpose personal computers, have separate processing and data storage components. This generalized design enables these systems to run a wide range of everyday applications. In these systems, energy efficiency and performance are bounded by the frequency of data movement between processors and off-chip memory [182].

This easily becomes a limiting factor with the increasing volume of data in massive-scale applications such as real-time sensor data and genetics data [42, 174, 188].

However, due to energy and bandwidth constraints and difficulty in scaling memory technologies, using large-size memory systems becomes infeasible [13, 49, 50, 168, 182, 215].

Moreover, the explosion of big data applications is driving the development of machine learning applications [203, 248]; this imposes severe challenges of data processing speed and scalability on conventional computer systems [54, 57]. Such applications require a stand-alone technical solution and computing platforms that are specifically designed for them.

These special-purpose computing systems, namely AI accelerators [54, 57, 58, 105, 259], usually require a deep understanding of the target workloads and are often constructed with a large number of highly parallel computing and storage units. Many of these systems are designed to be used

on the edge where data exists, therefore, they are constrained by limited resources and energy budget. Thus, similar to conventional computers, using large-size memory systems are infeasible in hardware in accelerators as well.

Over-provisioning of memory system resources alone cannot solve the rising demands of data processing and storage; in fact, efficient approaches for storing and processing data are critical in memory systems. Therefore, there is considerable amount of effort not only to improve the compute unit [18, 77] and memory technologies [75, 75, 175, 217, 270], but also to reduce the data movement [21, 200, 242, 243, 260] and therefore make this data movement cheaper throughout the memory hierarchy.

### 2.1.1 Multi-level Memory Hierarchy

Memory hierarchy separates computer storage into a hierarchy based on response time (i.e., latency) and capacity. Latency and capacity are related, as addressing a high-capacity memory incurs a higher latency.

An underlying assumption of the memory hierarchy is the data locality principle, consisting of temporal and spatial locality. Temporal data locality means that data that is accessed is likely to be accessed again in the near future, whereas spatial data locality means that data stored in adjacent memory blocks is likely to be accessed together.

Usually, when a piece of data is requested by the computing unit, it will be retrieved from the closest level to the processor. These memories are faster but have limited capacity. The request will be propagated further to the next levels if the data is missing at that level. Memories farther from compute unit have larger capacities, but take longer to access. Faster memories are vital for faster computation; thus, storing more data on levels closer to the processor will improve the system performance by reducing the access to slower memory levels.

While processor is executing instructions, random-access memories temporarily store data. Random-access memory cells are divided into SRAM (Static Random Access Memory) and DRAM (Dynamic Random Access Memory). Briefly, SRAM is fast but requires more silicon space, whereas DRAM is slower but allows for a higher capacity due to their simpler structure and higher density.

Designing for high performance requires considering the restrictions of the memory hierarchy, i.e., the size and capabilities of each memory. The typical memory hierarchy in both general-purpose and specific-purpose computing systems are as follows:

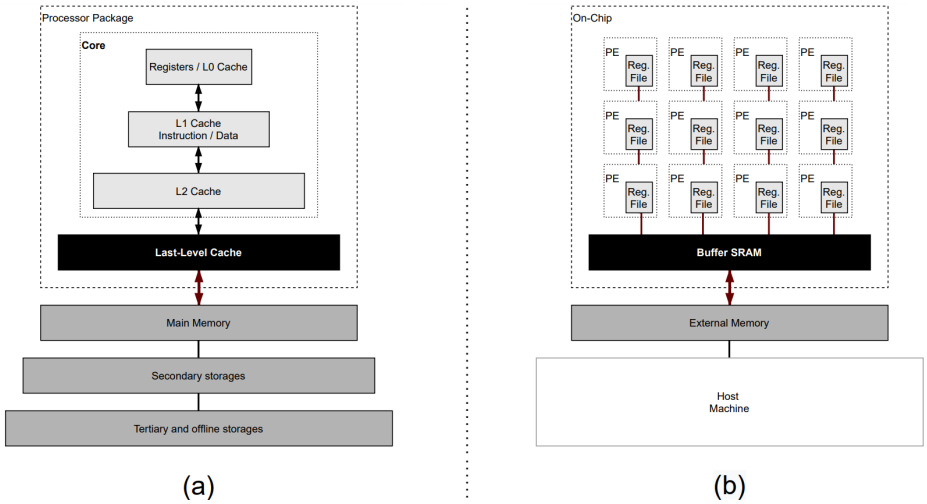



Figure 2.1: Typical memory hierarchy illustrated for (a) single-core general-purpose computing systems and for (b) special-purpose AI computing systems.

### General-purpose systems

Applications are becoming more sophisticated and as a result, their memory footprint keeps increasing [162]. In response, the number of levels in the memory hierarchy and the performance at each level have increased over time in the general-purpose computers. Latency, bandwidth, and capacity of each level are the key determinants of each level's performance.

Modern processors contain multi-level on-chip SRAM memories called cache memories, which leverage data locality in order to mitigate the latency and bandwidth limitations of accessing the main memory; therefore, levels closest to the CPU (i.e., registers and cache memories) focus on providing data with very low latency, constraining their physical size.

Moving away from the CPU, the latency cost of missing in a particular level of a hierarchy grows rapidly: from a few processor cycles for a first-level access, to hundreds of cycles for a main memory (i.e., DRAM) access, to millions of cycles for a disk access.

The memory hierarchy can be seen as a pyramid of storage mediums. Each computing system will have its specific design constraints and exact numbers. The following hierarchy (illustrated on Figure 2.1(a)) is what the typical memory hierarchy looks like in general-purpose computers these days:

- *Processor registers and Level-0 cache*: usually accessible in couple of cycles with a few KB of capacity.
- *Level-1 caches for instructions and data*: accessible in a few cycles with around hundreds of KB of capacity.
- *Level-2 cache*: with access times of a couple of tens of cycles with up to a couple of MB of capacity.
- *Last-level cache(s)*: shared storage with an order of magnitude higher capacities and access latency of tens of cycles.
- *Main memory*: resides off-chip and requires hundreds of cycles to access but offers GBs of capacities.
- *Secondary storage*: used to store bulk data with TBs of capacity and can be an order of magnitude slower to access. Examples are SSDs and HDDs.
- *Tertiary and offline storage*: can be an order of magnitude slower to access than secondary storage but offer orders of magnitude bigger capacity. Examples are optical disks and tapes.

Processors are typically running instructions on separate cores at the same time, increasing overall speed for programs that support multithreading or other parallel computing techniques. Each core has its own registers and first-level and second-level caches, but all cores share the Last-Level Cache (LLC) in current designs.

One of the major constraints with the increase in the number of on-chip cores is the tremendous increase of bandwidth requirements, especially off-chip bandwidth [182]. Off-chip bandwidth is mainly generated by the on-chip LLC misses and writebacks. These off-chip accesses depend on cache size and the cache replacement policy.

Improving cache memories can have a huge impact on overall computing system performance by avoiding the expensive access to off-chip memories.

#### special-purpose systems

Typical memory hierarchy of AI accelerators, as illustrated in Figure 2.1(b) is composed of an off-chip memory (e.g., DRAM) and multiple on-chip scratchpad memories. Compute efficiency and power consumption are major considerations especially in an edge deployment scenario.

A DNN generally requires a large memory footprint which affects both the energy and performance. The majority of the power consumption of these applications is due to the on-chip and off-chip memory accesses rather than compute unit. For large and complicated networks, it is unlikely that the whole network can be mapped onto the chip. Due to the limited off-chip bandwidth, it is important to reduce the off-chip data transfer to improve the computing efficiency. Moreover, the cost of accessing off-chip DRAM is orders of magnitude more than that of on-chip memory [58] which is another important consideration in reducing the off-chip access as much as possible.

There has been a lot of research on improving the memories in these computing systems in order to improve their performance and energy. Some [60, 75, 156, 175, 180, 217, 246] proposed emerging memory technologies as a solution. Although these may become very attractive as a possibility for future memory hierarchies, there are currently major issues with their reliability and performance, as described in chapter 1. In addition, the lack of compatibility of the technology with the fabrication process is also a very important issue [40].

Intel® Optane™ DC, the only commercially available device nowadays, suffers from lower bandwidth and higher latency compared to DRAM [130, 224], and targets the gap between DRAM and SSD memories. It is possible to achieve speedups using this device, but this requires redesigning the software specifically to access persistent memories directly [130].

## 2.2 Data Compression

Data accessed by real-world applications in the memory hierarchy of computing systems show a significant amount of redundancy, which provides an opportunity for compression [237, 243]. Such redundancy may arise due to the nature of program inputs and operations.

Several applications use a common value (e.g., zeros) for initializing a large array of integers that causes the most common data redundancy in applications. Zero is also used to represent null pointers or false boolean values, and to represent sparse matrices (in dense form).
Similarly, with variable assignment or on using *memcpy()*, two copies of data may be stored in the memory. A large contiguous region of memory may contain a single value repeated multiple times [223].

In several cases, programmers provision large-size data types to handle worst-case scenarios even if most values can fit in a smaller data type, for example, a 4-byte integer may store only a 1-byte value. Special cases such

as null values (i.e., all zero bits) can be represented with a single flag bit only. Further, in many cases, the differences between values stored within a memory block may be small, and hence they are occupying the memory space with repeated upper bits. A stride sequence might appear when a data structure such as an array is being accessed in a regular fashion; loop induction variables also have a stride characteristic [223].

Data compression is the process of encoding information using fewer bits than the original representation. Assuming that data consists of some number of symbols (i.e., bytes, words, etc.), compression methods replace these symbols with a new set of codewords that occupy less space in memory.

Data symbols can be replaced with codewords using two types of coding: a) fixed-length coding and b) variable-length coding. Fixed-length coding uses the same number of bits for each codeword. Variable-length coding can generate more efficient codes assuming that some of the symbols are more likely than others. By assigning shorter codewords to the more frequently occurring symbols and longer codewords to the symbols that occur infrequently, the average size of output codewords can be smaller.

Compression methods can also be categorized as static or dynamic. Static techniques provide a fixed mapping from data to the code words. Dynamic techniques can change the mapping over time and require only one pass on the input data to generate the encoded message on the fly.

Depending on whether the compression and decompression change the original information, data compression algorithms can be categorized as either lossless or lossy. In lossless compression methods, no information is lost and the size of data is reduced by identifying and eliminating statistical redundancy. On the other hand, lossy compression reduces bits by removing unnecessary or less important information. Therefore, there is a corresponding trade-off between preserving information and reducing size.

In the remainder of this section, we discuss common software-based lossless and lossy compression methods used for data storage and communication.

### 2.2.1 Lossless Data Compression

A lossless technique means that the restored data is identical to the original. This is absolutely necessary for many types of data such as executable codes. Lossless compression is possible because most real-world data exhibits statistical redundancy. For example, an image of snow may have areas of white color that do not change over several pixels. *Run-length encoding*, *delta compression*, *Huffman encoding*, and *dictionary coding* are among the

most popular algorithms for lossless storage. The main performance metric in these methods is compression ratio.

### Run-length encoding

Run-length coding is a well-known method based on the assumption of long data sequences without change of content. These sequences can be described by the value being repeated and length. For instance, a data block containing a value repeated eight times can be compressed to a block storing the repeated value itself followed by the value eight.

Therefore, run-length encoding is a good candidate for transmission of streaming data where the consecutive data blocks does not differ significantly from earlier data, but can be inefficient when there are lots of unique occurrences of data blocks. It is also used for compressing bitmapped images where many consecutive repeated values, for example, white pixel, might exist in the image. Image formats such as TGA [9] and BMP [7] are based on run-length encoding.

### Delta Compression

Delta compression is a way of storing or transmitting data in the form of differences (known as diffs or deltas) between sequential values rather than directly storing or transmitting the entire data. It is widely used in HTTP servers [222] to send updated web pages, to perform the online backup of user data with respect to the previous version [239], in the Git source control system to pack objects where the source or data files are changed incrementally between commits, to copy the data with lower bandwidth requirement using the rsync [8] tool, and so on.

Delta compression performs best when data has some small variation. Its effectiveness will be affected if it is applied on an unsorted data set where there is not much similarity among consecutive values. The decompression procedure also involves a sequential process depending on the delta calculation procedure.

### Huffman Coding

Huffman coding eliminates data redundancy using the statistical properties of data values. It is a static compression method and directly maps individual symbols to binary words. The Huffman coding algorithm is a simple and systematic way to design good variable-length codes given the probabilities of the symbols; it generates a tree by repeatedly joining two nodes with the

smallest probabilities to form a new node with the sum of the probabilities just joined. It assigns a 0 to one branch and a 1 to the other branch. The codeword for each symbol is given by the sequence of 0's and 1's starting from the root node and leading to the leaf node corresponding to the symbol.

Huffman coding [126] assumes the probabilities of the symbols are known apriori. This might not be possible for many applications where the data cannot be studied statically. Moreover, the value distribution of data being compressed is assumed to not changed (e.g., in the case of compressing files). To overcome these challenges, dynamic Huffman coding [247] which creates the tree as data enters at the cost of complicating the decompression procedure.

Despite achieving optimal compression rates for individual symbols, Huffman coding can be highly inefficient for compressing groups of symbols, particularly when the information content of each individual symbol in a sequence is close to zero. Moreover, it is possible for an infrequent symbol to have a wider Huffman codeword than its original representation [136]. This inefficiency can be caused by a bad selection of symbol granularity.

### Dictionary Coding

Dictionary coding works by going through a stream of symbols and substituting any sub-sequence which has already occurred in the stream with an index or pointer to the previous occurrence of that sub-sequence. If the length of the index or pointer is smaller than the length of the sub-sequence, then this results in savings. This is done either by creating a table or using sliding windows.

In methods that create a table, compression happens by analyzing the file to create a list of the $2^n$ most common words and then process the file from beginning to end based on that. Methods using sliding windows, create a search buffer and processing the file based on previous matches in that buffer. If the current word is in the dictionary/buffer, then it is replaced by a tag, followed by the position of the entry in the dictionary. If it is not available in the dictionary, then it is replaced by another tag followed by the word that remains unchanged.

The family of Lempel-Ziv(LZ) [285, 286] compression uses dictionary based techniques to perform dynamic compression of data. A variant which is a combination of LZ77 [285] and Huffman is called DEFLATE [2] and it is used in methods including ZIP [10], zlib [11], and gzip [3]. These schemes are generic and ideal for arbitrary collections of data, but at the same time very complex.

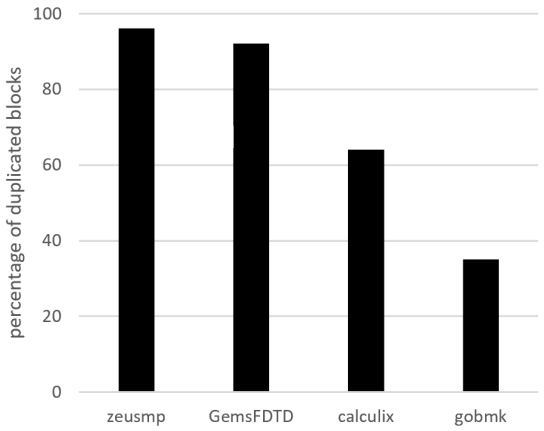



Figure 2.2: Percentage of the redundant cache blocks illustrated for several benchmarks from Figure 1 of [242].

### 2.2.2 Lossy Data Compression

In these schemes, some loss of information is accepted as dropping nonessential detail can save storage space. For example, an image of the sky may have variants of the blue color that are very similar to each other (but not identical) and the difference cannot be detected by eye if all are replaced with a single (similar) blue color. Most forms of lossy compression are based on transform coding which is widely used in multimedia formats such as in JPEG for images.

Another common way to compress data with loss of information is called quantization. Broadly, a quantization function maps the set of real values (full-precision data) to low-precision data with lower bit-widths. The function is chosen to minimize the quantization error between the full-precision values and their low-precision representations. This error is usually measured as the mean squared error (MSE).

As these methods trade off information to data size, both the error in data and the reduction in data size after compression (i.e., compression ratio) should be considered.

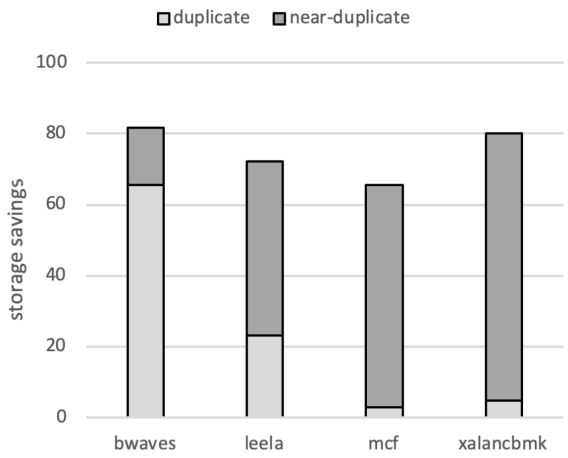



Figure 2.3: Potential storage savings when duplicate and near-duplicate data blocks are compressed on several benchmarks from SPEC2017.

## 2.3 Hardware-Based Data Compression

### 2.3.1 Opportunities

In previous sections, we talked about potential cases that can cause data redundancy. Now let us look into potential benefits if we avoid storing the redundant data by compressing on-chip data. Figure 2.2 depicts the percentage of the redundant data in cache memory for four benchmarks. In the case of zeusmp and GemsFDTD, the majority of the redundancy comes from null blocks (i.e., zero lines). Therefore, an ideal method that avoids storing null blocks can free up more than 90% of cache space in the case of zeusmp and GemsFDTD.

Figure 2.3 shows how much storage space can potentially be saved if we use a delta compression method to encode near-duplicate blocks as series of differences with respect to other residing blocks in cache. The figure also compares the amount of savings with respect to exact deduplication and highlights the huge potential of compressing near-duplicate blocks.

Aside from the data redundancy, in some special applications such as ones developed in AI field many works [48, 157, 161, 184, 229, 254] have claimed that a full-precision operation is not necessarily needed. These works propose various fixed-point systems with accuracies similar to the baseline accuracy to reduce the computational cost and storage requirements. Similarly, prior research [92, 107] verified that not all parameters of the neural

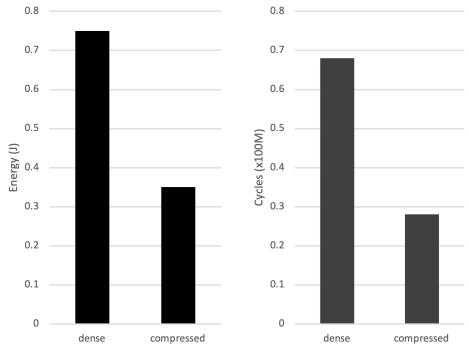



Figure 2.4: Potential energy savings and speedup from ideally leveraging all weight sparsity (here, 5× using [92]) in the forward path of training VGG-S (15M weights) to convergence.

networks are important, and the less important ones can be dropped from the computation and storage. For example, Figure 2.4 illustrates how a compressed model may lead to around 2 times energy reduction and speedup.

These insights about redundancies, inefficiency in data representation, and the possibility of using low-bit data types motivate the use of data compression in the hardware to make more efficient systems.

### 2.3.2 Software or Hardware Implementation

Compression performance is dependent on available computation resources [186] and the compression techniques employed. There is a trade-off between a compression technique that is implemented in software (i.e. mapping the compression algorithms onto dedicated processing cores) in comparison to a hardware compressor (i.e., application-specific compression logic embedded in the memory): at software level, there is more information about the data (e.g. related data) but less control over bit-level manipulations and data placement in the memory which is essential for high performance compression [288].

Several mechanisms were proposed to perform memory compression with software implementation for various modern operating systems in the compiler [151] or in the operating system [97, 250]. While using the additional information that software have can be quite efficient in reducing applications' memory footprint, the major limitation of these works is very slow decompression [199, 288]. Further, there is also a read latency overhead which comes from the fact that this way of decompression in software, consumes

CPU cycles [275] and extra CPU-DRAM data traffic.

Another difference between a software implementation and the way data is stored in the hardware is the block granularity. In filesystems that provide transparent compression (e.g., ZFS [39], Btrfs [208], and NTFS [210]), data compression is typically performed over fixed-size data chunks (e.g., 64 or 128 kB), and access to any 4 kB page requires reading and decompressing the entire chunk [51]. However, for example in the last-level caches, data are being stored typically in 64-byte chunks which are orders of magnitude smaller. Therefore, special considerations are needed to rearrange data in memories so that bigger chunks of compressible data can be compressed and placed together. Overall, these challenges require changing the file system or application layers both by modifying their standard data access behaviors [288] which imposes scaling challenges.

Aside from the block granularity, another fundamental difference is that software does not have direct access to the data being stored in several memories in memory hierarchy including on-chip cache memories which are hardware managed. This means, in order to have better data management in those memories, there is a need for special support and drivers, which adds new levels of indirection.

In addition, as software must keep tracking the location and size of each compressed data chunk (therefore, additional metadata look-up), the metadata management becomes more complicated. The variable sizes of compressed data can create inefficiencies such as adding another level of indirection, which slows down the whole compression and decompression processes. Slow compression and decompression can be tolerated for files stored in slower memories in the memory hierarchy (i.e. secondary storages) with low bandwidth and slow access times.

As a result, software-based compression is usually employed when the latency is not critical and storage capacity is the most important design goal; they are usually designed to provide better compressibility for a large amount of data whose size is several KBs and are not frequently accessed [288], therefore, more suitable for a secondary storage device [155].

Problems of software-compression are mostly avoided once compression is shifted down to the lowest level, i.e. by putting hardware compression engines into memories [282]. Hardware compressors have an order of magnitude higher data rates [16, 68, 240]. Moreover, valuable CPU bandwidth can be freed by offloading the compression task to these specialized hardware which can also reduce the power consumption [16], as a result, CPU resources can be allocated to run other tasks in parallel.

### 2.3.3 Challenges of Hardware Implementation

We briefly listed several major considerations for practical implementations of compression in hardware in chapter 1. Here we describe them in more detail:

- Reasonable complexity and implementation cost
- Minimum overheads
- Zero or low application quality loss

A main challenge to implement any compression method in hardware is to balance the compression ratio and hardware simplicity. The conventional way to achieve this is usually to aim for the highest possible compression ratio (i.e., using existing software-based compression algorithms) and then simplifying these algorithms so that they can be implemented in hardware. An alternative option is to prioritize the simplicity of the compression algorithm over its compression ratio and choose simpler methods.

Another major challenge is accounting for the overheads, for example, the effect of a compression method on the latency of the system. As processor performance is sensitive to memory access latency, it is critical that the decompression latency be as small as possible when accessing any compressed data in the memory hierarchy; otherwise, this may hinder the system performance. Using compression methods in on-chip memories with a few cycle budgets means that every added cycle can have an impact on system performance and should be considered while designing or choosing the compression method. The same considerations are needed for the area and energy overheads.

Moreover, compressing data into variable-size blocks as well. poses several challenges, including: a) where to store these variable-size blocks in the memory, b) how to avoid the memory fragmentation, and c) how to later locate these blocks in the memory.

One design consideration in compressed memories is how to store variable-size blocks. For example, one can store the same-size blocks in the same structure/region of the memory for lower hardware complexity as the boundaries of each block are known on that region. Another design can place all the blocks with different sizes in a unified regime to make the most use of the available space. This greatly depends on the application the compression is applied to and should be studied carefully.

Another challenge posed by the variable size blocks is data fragmentation. The problem arises when an already stored compressed data block is

updated with new data with a different compressed size. A bigger new block means older blocks should be evicted and a new memory space should be allocated for the new block. This can leave some of the old space empty (cause fragmentation). Similarly, if the new compressed block is smaller than the old one, one option is to store this compressed block in the same old space and again end up with an empty leftover space. In naive implementations, this could lead to significant energy waste and design complexity due to shuffling data around.

Locating a variable size compressed block in the memory is also another challenge. In uncompressed memory, finding a certain cache block is usually trivial and calculated from the address/offset of the block being accessed. As data is not stored in the conventional manner in the compressed memories, there is another level of indirection (e.g., added offset or pointer) in order to locate the data block. This usually means there is a need to repeatedly recompute this new offset (or pointer), or store it somewhere in the memory. If not done carefully, this can lead to significant latency overheads and considerable design complications.

Lastly, the effect of compressing data on final application quality should also be considered. Most applications do not tolerate any errors, therefore, need lossless compression methods, which can limit the amount of compression possible. In some cases, such as machine learning applications, some loss of accuracy is acceptable for bigger compression gains. These lossy methods should be carefully chosen and sometimes tune to make sure the amount of quality loss is minimal.

### 2.3.4 Inefficiencies of Using Traditional Methods

As we mentioned, the performance of the compression is not only dependant on the resource that runs it, but also depends on the algorithm that is being performed. Most software-based compression algorithms are not suitable for low-latency hardware implementation due to their high complexity. They usually assume files are stored on disk, and their content will be intact during and even after the compression process.

In contrast to accessing files from disk, three things are fundamentally different with the data stored on-chip:

- Data is commonly accessed randomly, rather than sequentially
- Latency and also most of the time, energy are extremely critical
- File content does not change after compression

In traditional methods, files are often accessed as sequential streams, and the large decompression latencies are considered to be acceptable given that disk accesses are already slow.

Lempel-Ziv algorithms need to trade-off compression latency with compression ratio by tuning the sizes of sliding windows. Prior attempts to use the optimized versions of the Lempel-Ziv algorithm [15] in hardware had decompression latencies of more than 60 cycles, which makes it not suitable for on-chip memories with 1-10 cycles of access time. One way to reduce the time complexity, is to analyze the file statically and come up with some predefined items in dictionary. Also, some variants of Lempel-Ziv [285] needs to scan through the entire file, building up a dictionary of common character sequences before the compression pass.

In the static Huffman algorithm, the frequency of the symbols should be known beforehand or should be calculated by looking at the entire data once to create the encoding for the Huffman tree stored in the frequency table. The problem arises with the table size: a larger frequency table gives a better compression factor but reduces speed, increases area and energy consumption in a hardware implementation [136]. Besides, applying static Huffman to compress the data during the execution of a program, may degrade the compression ratio to unacceptable low levels [136]. The same overhead exists with the dynamic version of this method. Dynamic Huffman is even more complex due to the effort to form the tree dynamically and requires more area and energy to operate [247].

Moreover, the fact that some memories in the memory hierarchy (e.g., on-chip caches and main memory) are accessed randomly creates additional challenges including efficiently locating and decompressing arbitrary blocks of potentially variable sizes.

Lastly, the data stored in files does not change after compression is done. On the other hand, data stored in memory hierarchy, more specifically on-chip memories, will constantly change upon every insertion and eviction to and from that memory. Similarly, writes to any block that changes any word of that block also require that the entire block be compressed again and potentially produce a new uncompressed or compressed data block. This requires in-hardware compression methods that can dynamically handle these changes and still efficiently compress the data.

Therefore, directly applying well-known compression algorithms which are usually implemented in software, leads to high hardware complexity and unacceptable decompression/compression latencies, which in turn can negatively affect performance.

| | Off-Chip DRAM | SRAM Buffer ($> 100KB$) | Register File (0.5KB) |
|---|---|---|---|
| Normalized Energy | 200× | 6× | 1× |

Table 2.1: Cost of accessing off-chip and on-chip memories in 65nm normalized to a MAC operation (Table IV in [57])

## 2.4 Hardware-Based On-Chip Memory Compression

Accessing the off-chip memories is orders of magnitude more energy costly and considerably slower than on-chip memories (See Table 2.1); thus, it is preferred to have larger on-chip memories to avoid these costly accesses. Unfortunately, adding more on-chip silicon area is not an option. recalling from Chapter 1, increasing the size of the on-chip memory not only increases the area and leakage power, but also hurts the dynamic energy and the access time; this makes larger memories challenging to be used in latency and/or energy critical applications. In fact, in current multi-core systems, the last-level cache capacity per logical core tends to be less than only 1MB [119]. Therefore, any optimization that results in increased effective capacity of on-chip memories, rather than physical memory space, has a great impact on energy and performance of the entire system.

*Memory compression* is an effective technique to increase on-chip memory capacity as well to decrease the on-chip and off-chip bandwidth usage. Hence, there is a need for a simple yet efficient compression technique that can effectively compress common data patterns, and has a minimal effect on memory access latency.

Compression methods for on-chip memories deals with frequently accessed data whose size is only several bytes (e.g. 64B) as compared to several kilo-bytes in software based methods [288]. Some of the drawbacks of software-compression such as high decompression latency as mentioned in the previous section, makes using the software-based implementations and algorithm impractical with the tighter latency budget, smaller data block granularities, and high bandwidth requirements of on-chip memories.

On the other hand, specific compression schemes are more suitable to compress the data in on-chip memories as compared with the traditional generic data compression methods. However, there are several inefficiencies of performing application specific compression techniques in software;

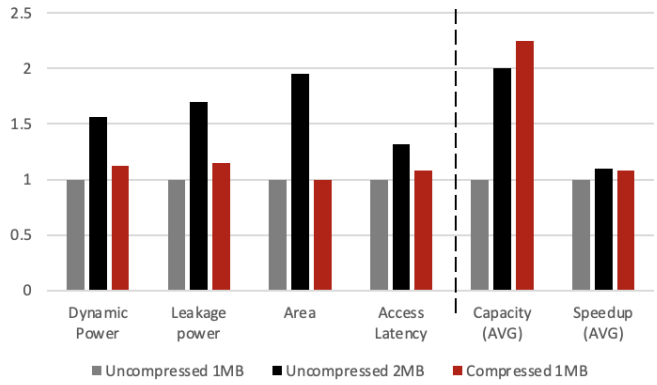



Figure 2.5: Comparison of dynamic read power, leakage power, silicon area and access latency when compressing the LLC (compressed 1MB) as opposed to doubling the size of the cache. All values are normalized to the 1MB uncompressed cache. Capacity indicates the effective capacity of each cache. For speedup, only the workloads that benefit from compression are considered (details in chapter 4).

these algorithms typically feature complex, scheme-specific bit manipulations [21, 143, 200] and variable control, both of which are ill-suited to the wide, deep processing pipelines found in modern CPUs [142]. As a result, application-specific compression schemes that require high performance should be implemented in hardware.

Figure 2.5 illustrates how a hardware-based compressed 1MB cache (described in chapter 4) can fit more than double the cachelines as the conventional uncompressed 1MB cache and achieve similar speedups of uncompressed 2MB cache. This compression technique requires only 1% of uncompressed 1MB cache silicon area for implementing the compression logic. Therefore, doubling the effective capacity in this way, does not incur the area, latency, and power overheads of a uncompressed 2MB cache.

We continue this section by providing a brief background on on-chip cache and scratchpad memories, and discussing existing compression proposals.

### 2.4.1 Compression in Caches

As mentioned earlier in this chapter, memory hierarchy consists of multiple cache memories with the smaller L1 cache being closer to the CPU and the

bigger LLC being farthest. This way, more frequently needed addresses will be accessed at the highest speeds. Cache compression can improve cache hit rates by fitting more blocks in the cache, provided the blocks are compressible.

One question that arises is: which of these caches are better targets for compression? To answer this, we need to consider what are the challenges of compressing each cache level and what will be the potential benefits.

Unlike compression, which takes place in the background upon a cache fill, cache decompression is on the critical path of a cache hit access. Choosing a cache close to CPU as the compression target, like the L1 cache, brings mediocre benefits: first, L1 cache hit times are of utmost importance, therefore, there is a very tight cycle budget for decompression task before it starts hurting the performance. Trying to constrain the method for faster decompression means that only simple methods can be leveraged, limiting the compression method's storage savings. Second, the timing improvement per cost of added overhead can be questionable, as it only reduces the access to the next on-chip cache level (which is only a few cycles extra). Finally, the amount of data stored at that level is pretty low, which means there is less chance for compression to leverage spatial locality and redundant data, and as a result less storage saving benefits.

LLCs on the other hand, have larger capacities, which can result in capturing more data locality and considerable storage savings. They have more cycle budget for decompression task to be performed as well; this means not only that the compression method is not limited to only very simple ones, but also that the penalty accessing the compressed line is low given the original access latency. Finally and most importantly, improving LLC will avoid the costly off-chip accesses, saving 100s of cycles on every cache hit due to the increased effective capacity.

To better understand how data is stored and accessed in LLC, we need to look into the LLC organization. The LLC logically consists of several sets and each set contains multiple ways; modern LLCs have 4–16 ways per set. Physically, the LLC consists of tag and data arrays, usually with a dedicated tag and data array for each way in a set: e.g., an 8-way LLC has eight tag and eight data arrays as illustrated in Figure 2.6. Each cacheline in the data array is allocated one tag in its respective tag array: when an incoming memory block is placed in the LLC, it is assigned to the set that corresponds to its address, and replaces the tag and data entries for one of the ways. If this way previously contained valid data, this data is first evicted.

The capacity of a cache has a major impact on performance, die area, and power consumption. The decision of how large to make a given cache

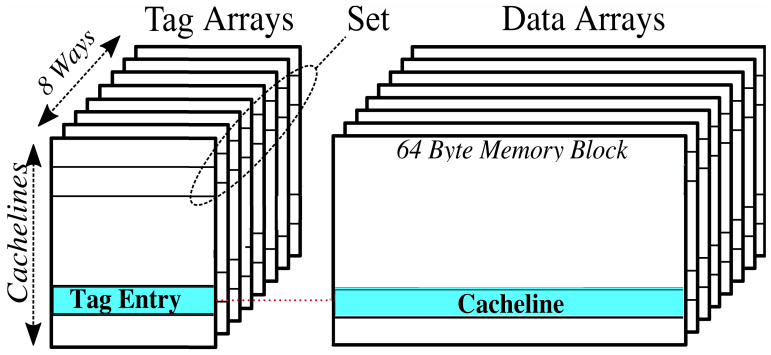



Figure 2.6: Structure of a 8-way set associative cache.

involves trade-offs: while larger caches often result in fewer cache misses, this potential benefit comes at the cost of a longer access latency and increased area and power consumption. Therefore the caches are limited in their sizes and adding more silicon area is not a desirable way of increasing cache capacity.

Several prior works [21, 78, 104, 191, 200, 242, 250, 260] have tried to reduce cache misses, and improve the bandwidth requirements through cache compression. This requires several changes to the cache organization, such as resizing the size of arrays and changing the way the data lookup is performed.

In order to store more data in the cache via compression, the LLC needs to have multiple tags per cacheline. One way to do this is to increase the associativity of the tag array (e.g., double it) while keeping the associativity of the data array intact. For example, Figure 2.8 shows a structure used by [200] to store up to two memory blocks per cacheline. An alternative is increasing the number of tags and decoupling the tag array and data array so that multiple tags can point to any data entry in data array [242].

In general, cache compression proposals take advantage of *either* of two different dimensions of redundancy in cached data:

1. *Inter-block redundancy* exists when multiple cache indices store the same blocks of data. This can result from symmetry of some kind (e.g., fluid flow around a symmetric object), when sizable parts of the working set have the same value (e.g., the background of an image), etc. In this scheme, each incoming memory block is examined in isolation, and, if possible, compressed independently of other blocks.

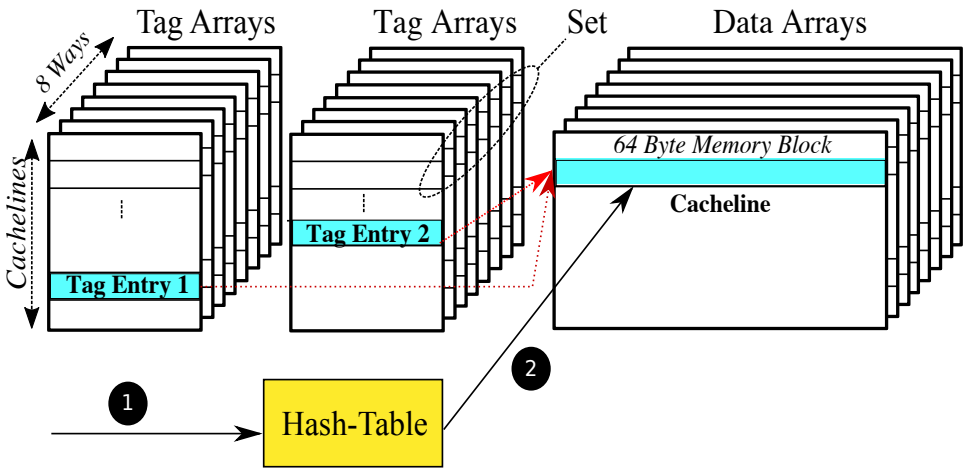



Figure 2.7: Last-Level Cache (LLC) implementing Deduplication (Dedup) [242]. Dedup enables multiple tags to point to the same cacheline containing a common memory block. During the LLC insertion of the memory block, Dedup uses a hash-table of the most recently used data values to identify exactly identical cachelines.

2. *Intra-block redundancy* exists when a single cache block contains compressible patterns within itself. For example, integers are usually allocated at 32-bit or 64-bit sizes but their values often fit in the least significant byte; similarly, pointers used in a data structure may have been allocated close by and so may have identical most significant bits.

### Inter-Block Data Compression

Block-level data deduplication techniques leverage the observation that many cache blocks are either entirely zero [76, 78, 200] or are copies of other blocks that concurrently reside in the cache [59, 70, 117, 231, 242]. Instead of storing several identical copies, they aim to store only one copy of the block in the cache, and propose techniques to point the redundant data entries to this single copy.

To address inter-block redundancy, Dedup [242] modified a conventional cache to store one copy of the redundant data and reference it with multiple tags. The tag array is still arranged in sets and ways; the data array is decoupled from the tag array, and is designed to be explicitly accessed by pointers. An augmented hashing technique is used for faster duplication detection. Thereafter, a quick look-up occurs in the hash table indexed by

the hashed data.

### Intra-Block Data Compression

A simple technique to increase the capacity of the LLC is to employ intra-cacheline compression. For some applications, data values stored within a block have a low dynamic range resulting in redundancies [21, 200, 250]. Prior work [200] categorized these into (a) repeated values, (b) a set of values (especially zeros) repeated in a data block, and (c) near values, a set of values with the same upper data bits and different lower bits.

One way to reduce redundancy within the memory block is to capture the replicated data in dictionary entries and then point to that entry when new replicated data is presented [21, 129, 250].

Another method to reduce redundancy of nearly identical values is to try to separate repeated parts of values from distinct lower bits in a memory block. [191] extracts distinct 4-byte chunks of a memory block and uses encoding schemes to compress them, with dictionaries potentially shared among contiguous blocks.

Base-Delta-Immediate (B$\Delta$I), a state-of-the-art intra-cacheline LLC compression technique [200], exploits the insight that, in many workloads, data values within a memory block are similar, and therefore can be compressed as a "base" value combined with small offsets. To store up to two memory blocks per cacheline, B$\Delta$I doubles the number of tag arrays for each way in the LLC, as shown in Figure 2.8.

Figure 2.9 shows an example of this process. The block *d1* consists of 64-bit floating-point numbers whose values are close. The intra-block compression reduces the block to 40 bytes (an 8-byte base value followed by eight 4-byte offsets), and compacts it to take up 5 segments in the set.

**Limitations**: Intra-cacheline compression schemes can work well when the working set consists of arrays of primitive data types with a relatively low range of values. However, they do not capture the structural properties of more substantial, heterogeneous data structures, whose redundancy surfaces only when considering multiple cachelines.

Moreover, aside from B$\Delta$I, none of the existing methods account for the near-duplicate similarity in the data stored on-chip and miss a huge potential in reducing the memory footprint. Proposals like exact deduplication can only exploit data regularity if multiple LLC lines have *exact* data values [242], while techniques that directly leverage program-level data structure information require pervasive program changes and ISA extensions [243]. Some work like [21] suffers from high decompression latency

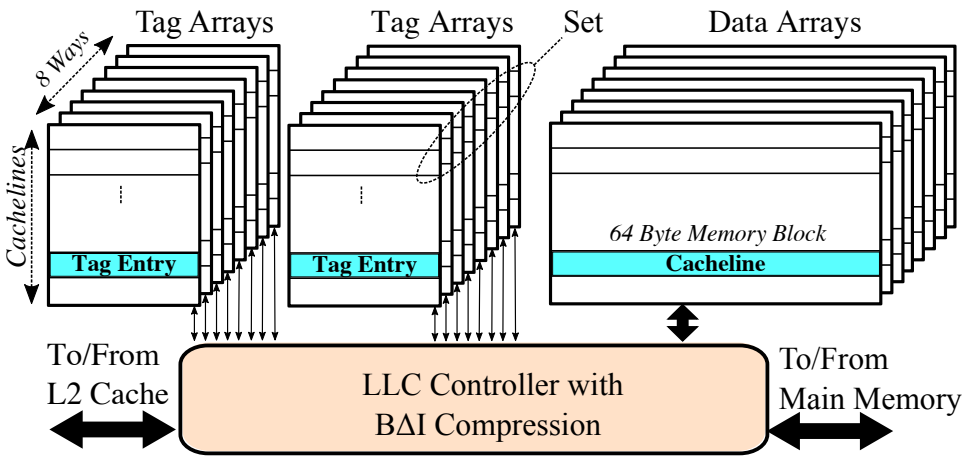



Figure 2.8: Last-Level Cache (LLC) implementing BΔI compression. The LLC has multiple tags per cacheline to store additional tags for each of the compressed memory block within the physical cacheline.

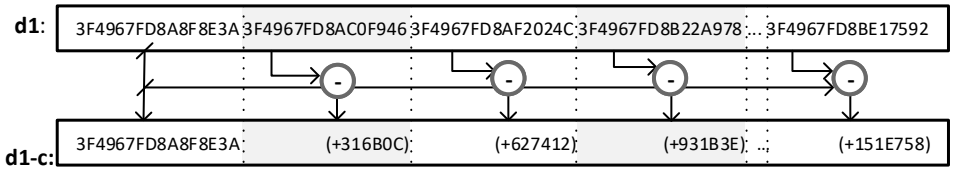



Figure 2.9: Example of compressing a 64-byte cache block from roms_r benchmark (*d1*) using BΔI. The block consists of 64-bit floating-point numbers whose values are close; they are compressed to a 64-bit base value followed by eight 32-bit offsets, for a total compressed size of 36 bytes.

which can affect the system performance, while other proposals cannot achieve significant compression ratios [200, 242] and therefore speedups. Some cache compression methods that return approximate values [173, 211] work well for noise-resilient data (e.g., images), but are unsuitable for general-purpose workloads.

### 2.4.2 Compression in Scratchpads

DNNs have been applied to different applications and achieved dramatic accuracy improvements in many tasks in recent years. These works rely on deep networks with millions or even billions of parameters, and megabytes to gigabytes of footprint.

Reducing off-chip memory accesses in AI applications saves a lot of energy [58, 105]. Scratchpads are high-speed memories used to hold data for rapid retrieval and are explicitly manipulated by applications. Therefore, there has been a lot of focus to improve the on-chip memory requirements and footprint. Broadly, these works propose to employ novel methodologies to make machine learning models more efficient. Efficient models are models with compressed values (i.e., weights, activation maps, etc.) and similar performance (i.e., accuracy) having less computation and memory demand. For the purpose of this thesis, we only focus on the two common techniques, pruning and quantization.

#### Model Compression

Model (weight) compression techniques, which the majority of the DNN compression proposals focused on [29, 30, 33, 67, 81, 94, 102, 107, 115, 116, 122, 131, 135, 146, 177, 234, 251, 274], can be divided into the following general categories:

- Pruning: explores the redundancy in the model weights and try to remove the redundant and uncritical ones. This makes the model small yet sparse, thus reducing the off-chip memory access [67, 107, 122, 177].
- Quantization: allows the model to operate in a low-precision mode, thus reducing the required storage capacity and computational cost [33, 81, 94, 131, 146].
- Low rank factorization: uses tensor decomposition to estimate the informative weights of the DNNs [135, 234, 251].

- Transferred convolutional filters: designs special structural convolutional filters to reduce the weight space and save storage and/or computation [29, 30, 102, 274].

- Knowledge distillation: learn a distilled model and train a more compact neural network to reproduce the output of a larger network [115, 116].

Early works on model pruning showed that it is effective in reducing network complexity and addressing the over-fitting problem [96]. The Optimal Brain Damage [20] and the Optimal Brain Surgeon [21] methods, reduced the number of connections based on the Hessian of the loss function. A following trend in this direction is to prune redundant, non-informative weights in a pre-trained DNN model [106].

Prior work on DNN inference has established that it is unnecessary to represent data — both model weights and model activations — at full (32-bit) precision [94, 131, 146]. Often, instance, bit-widths can be reduced from 32-bit to 8-bit precision (reducing the memory footprint by 4×) while incurring a top-1 accuracy drop of under 1% [96] on large-scale datasets such as ImageNet [209].

**Limitations**: Pruning has become common practice to make compressed sparse networks but one major issue those proposals are facing is their need to train a network, prune, and then retrain to get the efficient sparse network [92]. This means that they will see a negligible memory and computation savings over course of training a network. While this saves energy at inference time, training the pruned network takes *more* time and energy than training an equivalent dense network to the same accuracy. Skipping the pre-training step is not an option: even if oracular knowledge of the pruned model connectivity is assumed, training the pruned model from scratch sacrifices accuracy compared to the original network [107, 159].

Several works [179, 278] achieve only small pruning factors and suffer accuracy loss. Some [84, 278] prune the model very gradually; this implies (i) no peak memory footprint reduction, (ii) mediocre energy savings because the average sparsity is low during most of the training process. The remaining technique [92] maintains the target weight sparsity throughout training, but gives up computation sparsity — a significant drawback for training, where weights are usually 32-bit floating-point numbers that are energetically expensive to multiply.

The issue with the existing quantization based methods is that typically they cannot achieve substantial memory savings without losing too much

accuracy. Methods that quantize models after training, suffer from this more severely. To compensate for this accuracy loss, several works proposed to perform the quantization while training the network which again hinders training-time storage savings. Ideally, we want a mechanism that has the benefits of the efficient model from the beginning of the training task.

Finally, the majority of the DNN compression mechanisms miss compression opportunities as they focus on the weight compression, however, the activation maps have become the dominant data type in the modern compact networks like ShuffleNetV2 [281] and MobileNetV2 [216]. Compression techniques are more difficult to apply to activations because activations change for every input while weights stay constant once the model is trained. Activation maps must be compressed dynamically, so the model cannot be retrained offline to reverse the accuracy drop due to activation quantization, as is commonly done with weight quantization [61, 106].

### Activation Compression

DNNs not only can be accelerated by compressing weights, but also by making the activation maps efficient. Unlike weights that are trainable parameters and can be easily adapted during training, activation maps need special consideration as they are input dependant and are generated dynamically at the runtime.

Activation map size plays an important role in power consumption and memory footprint of DNNs [58, 105]. In low-powered AI accelerators, on-chip memory is extremely limited [54, 58, 105]. The accelerator is bound to access both weights and activations from off-chip DRAM, which requires approximately 2 orders of magnitude [58] more power than on-chip access, unless the input/output activation maps and weights of the layer fit in the on-chip memory.

Most modern Convolutional Neural Networks (CNNs) use the Rectified Linear Unit (ReLU) as an activation function. As a result, a large percentage of activations are zero and can be safely skipped in multiplications without any loss. One issue with pruning is that it introduces imbalance in the working set, and so, compression may not translate directly to faster inference since modern hardware exploits regularities in computation for high throughput.

Zero-skipping input activation maps to save computation and storage is widely used in AI accelerators [23, 105, 193]. A network would greatly benefit in terms of memory usage and model acceleration by effectively sparsifying even more activation maps. One drawback for pruning meth-

ods is that they usually requires long fine-tuning times that may exceed the original network training, for example by a factor of 3 or larger [177] in the case of EIE [105]. There has been some work to prune the activation channels [52] or to make activation maps sparser [87] using regularization terms. Regularization-based pruning techniques require per layer sensitivity analysis which adds extra computations.

Similar to quantization for model weights, there has been some work on quantizing the activation maps, such as [48, 96]. These methods usually suffer from huge accuracy drops in low bit-widths and need retraining before they can be used.

**Limitation**: A drawback which is more prominent in compressing the activation maps of networks is that reducing the bitwidth of the data to very low bit-widths (i.e., below 6 bits) to date resulted in significant accuracy degradation [125, 229], limiting the benefits.

Methods like quantization-aware training [61, 95] and fine-tuning [106] try to mitigate the accuracy drop due to activation quantization during the training phase, by calibrating quantization ranges on different inputs. However, they generally require access to at least some of the training data as well as the details of the training procedure (hyperparameters etc.), something often impossible with commercial vendors on account of security, privacy, or trade secret concerns.

To address this challenge, recent work has proposed post-training quantization (PTQ) techniques that do not require access to the original training data [48, 164]. While these PTQ techniques are effective in quantizing activations down to 8 bits, they fail dramatically at lower bit-widths. Figure 2.10 shows the impact of low-bitwidth activation maps on the top-1 model accuracy using the best state-of-the-art method [48].

### 2.4.3 Addressing the Limitations of Prior Methods

Existing compression methods suffer from either a small compression ratio, high hardware complexity, large decompression latency, or unacceptable quality losses. These limitations restrict the expected benefits from the compression task. There are lots of applications that are already increasing in size [38, 43, 114] and contain considerable amount of redundant and similar data that prior methods are not leveraging.

**We capture larger amount of on-chip data**: Throughout this dissertation, we develop novel compression techniques for on-chip data to further reduce the memory footprint of workloads by considering and compressing

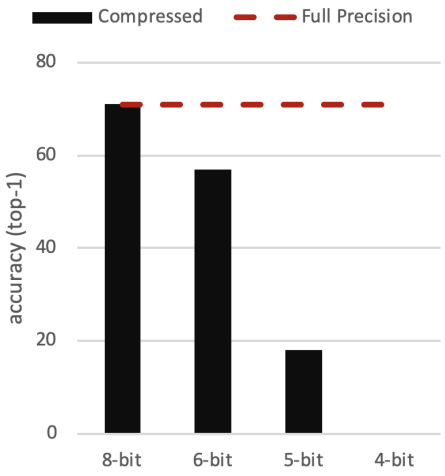



Figure 2.10: Impact of best state-of-the-art activation value quantization method [48] on the top-1 inference accuracy of the MnasNet1_0 network.

a larger amount of on-chip data that are dynamic and constantly changing. This, essentially enables the compression method to capture more relevant data and make better compression decisions.

**We leverage data similarity, not just redundancy**: We observe that there are similarities among data stored in on-chip memories caches and scratchpads, which we leverage towards developing efficient compression methods.

**We use dynamic clustering**: We group data by searching the entire on-chip memory and clustering or partitioning. The grouping is performed based on statistical properties, similarity, and importance of data. Then we apply specific compression to each group.

**We utilize hardware-friendly algorithms**: In order to convert these compression methods to on-chip savings, we use hardware-efficient variants of sorting and dynamic clustering methods.

# Chapter 3

# Leveraging Redundancy Within and Across Blocks

In this chapter[1], we argue that in order to develop an efficient compression method, it is essential to consider data redundancy both across and within data blocks. In general, redundancy in workloads varies widely. Some benchmarks have only intra-block redundancy, some only inter-block redundancy, and there are several workloads that showcase both types of redundancy. Moreover, we show that caches can be comprise a lightweight decoupled approximation stage to reduce the redundancy in data.

To this end, we propose an in-hardware compression technique, 2DCC, that leverages both the inter and intra block redundancies: it allows working sets that contain *either* type of redundancy to be compressed while also enabling compressing working sets that contain *both* types of redundancy. We also demonstrate that approximation and compression are orthogonal, complementary techniques that should be decoupled in cache designs and propose a cache compression method for approximate data.

First, we illustrate the opportunity in the state-of-the-art cache compression methods missed by not capturing the data redundancy in various granularities dynamically. Next, we overview the proposed cache architecture and describe the cache operations needed when a new line is inserted and evicted from the cache, followed by the new structures and operations added to support both types of redundancies in 2DCC. Because this requires decoupling cache structures, replacement policies become a challenge. 2DCC therefore uses separate replacement policies for the tag array and the data array, which optimizes for both reuse and space savings. Then, we study extensive set of benchmarks, categorize them based on their cache footprint, and evaluate 2DCC on them. When applied to the LLC in a server-class CPU, compared with best prior methods that compressed the cache to only 1.43×–1.49×, 2DCC achieves 2.12× geomean compression factor

[1]Parts of this chapter appear as: A. Ghasemazar, M. Ewais, P. Nair, M. Lis, "2DCC: Cache Compression in Two Dimensions," In DATE, 2020.

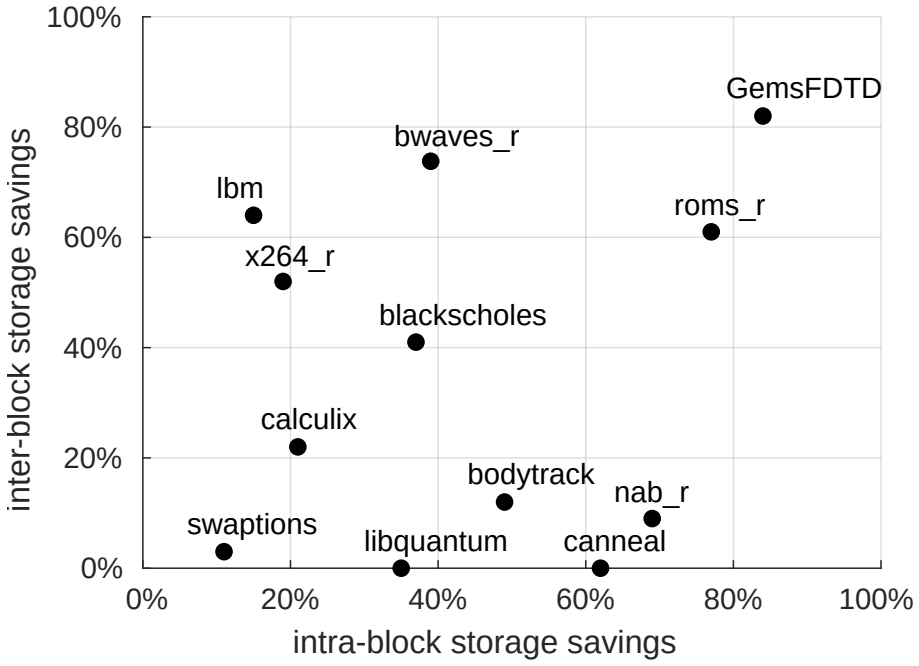



Figure 3.1: Space reduction in last-level cache images (100 per benchmark) using entropy encoding (Huffman compression). The x-axis shows encoding within each cache block (symbol size of 1 byte); the y-axis shows encoding across cache blocks (symbol size 64 bytes). 0% indicates that no compression was possible while 100% would indicate that the entire cache was completely compressible. These results provide a motivation for 2D cache compression.

across cache sensitive subset of SPEC CPU2017 [14], SPEC CPU2006 [114], and PARSEC [38]. 2DCC resulting in a geomean 11.7% speedup compared with best prior methods given the same silicon budget. Finally, we propose an approximate cache compression mechanism where approximation and compression stages are decoupled, and evaluate it on extensive set of benchmarks. When applied to LLC, it achieves better compression than bespoke approximate caches.

## 3.1 Beyond Single Type of Redundancy

As discussed in chapter 2, state-of-the-art cache compression methods [21, 78, 104, 191, 200, 242, 250, 260] have tried to increase effective cache capacity by focusing on either inter-block or intra-block data redundancy, however, real workloads exhibit a wide variety of redundancy patterns. To demonstrate this, we estimated intra-block and inter-block entropy in last-level cache snapshots from a range of SPEC [114] and PARSEC [38] benchmarks (see section 5.6 for details) by using Huffman compression [126]. To estimate inter-block entropy, we compressed the entire cache using 64-byte symbols (i.e., one cache block); to estimate intra-block entropy, we compressed each block independently using one-byte symbols.

Figure 3.1 shows how much space can be recovered for each benchmark by taking advantage of inter-block entropy (y-axis) and intra-block entropy (x-axis). Some benchmarks show significant savings by using only one type of redundancy: for example, *lbm* has many identical blocks which are generally not amenable to intra-block compression, while the blocks cached by *canneal* have intra-block value redundancy but most cache blocks are different. Others, such as *bwaves* and *roms*, contain a mixture of identical blocks and some compressible blocks. (The outlier, *GemsFDTD*, has nearly all of its working set filled with zeros and is therefore trivially compressible.)

For example, Figure 3.2(a) shows cache block fragments of last-level cache snapshots for three benchmarks, along with the number of exact copies of each block found in the cache.

The top panel shows three blocks of the destination grid written inside *LBM_performStreamCollideTRT()* in *lbm_r*. The cache block is filled with 64-bit floats, which differ enough that intra-block compression (e.g., B$\Delta$I [200]) is ineffective. Because of fluid flow symmetry, there are multiple copies of many cache blocks, all of which can potentially be deduplicated, allowing the size of the cache snapshot to be reduced by 67%.

The middle panel shows three blocks addressed by *swap_locations()* in *canneal*. In contrast to *lbm_r*, the working set contains no duplicate blocks. However, there is substantial intra-block redundancy: the data consists mainly of small 32-bit integers (netlist elements and locations). This allows the cache snapshot size to be reduced by 61%.

Finally, the third panel shows cache blocks from *roms_r*, an ocean forecasting model. The locality of behaviour within an surface patch, together with similarities across some patches, creates both intra-block and inter-block redundancy: many cache blocks in the working set are present in several copies, and each contains 64-bit floats that are close to one another.

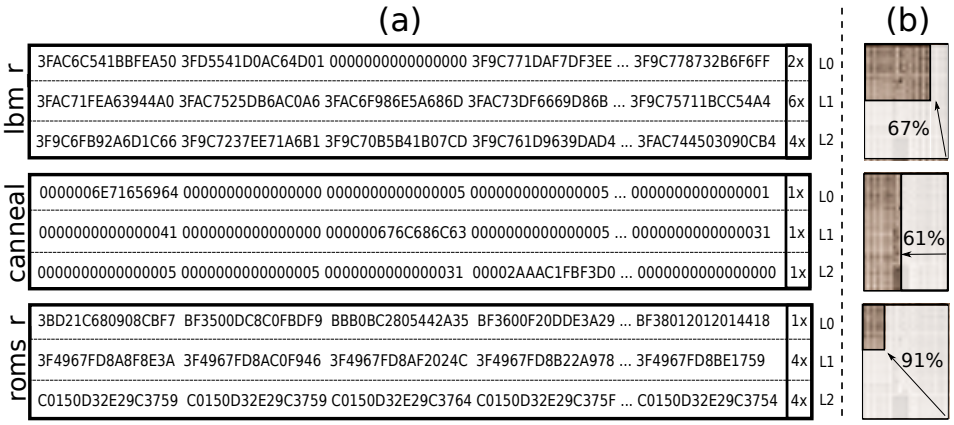



Figure 3.2: (a) Redundancy in LLC snapshots of three benchmarks: *lbm_r* shows inter-block redundancy: the three cache lines shown appear twice, 6 times, and 4 times; *canneal* shows different blocks each of which has a compressible 0 prefix; in *roms_r*, blocks appear in multiple copies but words also have similar prefixes. (b) Cache space saved using inter-block compression (y-axis) as well intra-block compression (x-axis) for these LLC samples.

For these cache snapshots, taking advantage of both forms of entropy can potentially save 91% of the cache space.

Figure 3.2(b) shows the potential cache silicon savings for intra-block (x-axis) and inter-block (y-axis) entropy by using an ideal compression method on the cache snapshots analyzed.

## 3.2 Architecture and Operation

Unlike conventional caches, which store one full (e.g., 64-byte) data block for every tag, compressed caches can either store multiple blocks in the same space [20, 21, 200, 260] or store only one block for multiple tags [242]; 2DCC similarly decouples the tag and data arrays. In contrast to prior approaches, each tag may point to an 8-byte segment anywhere in the data array rather than to only one index or a few possible locations; this maximizes data array utilization. To avoid storing duplicate blocks, multiple tags may point to the same segment. To detect inter-block redundancy, 2DCC adds a third structure — the *hash array* — which stores summaries of cached blocks and allows the controller to quickly identify duplicate lines.

When a 2DCC cache inserts a new block, it checks whether an identical

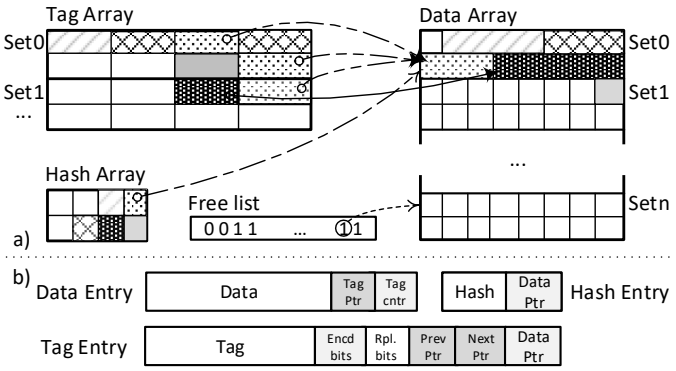



Figure 3.3: (a) The three decoupled storage structures that comprise the 2DCC cache: arrows show pointers logically linking the structures. (b) Entry contents.

block is already present; if the block is a duplicate, then a reference to the existing block is inserted instead. If the block is unique, 2DCC attempts to compress it and store it in a part of a line in the cache's data array, with the rest of the line usable by other compressed cache blocks.

This approach presents several challenges. Firstly, duplicate cache blocks must be detected quickly. Secondly, allocating and evicting blocks with different compression factors must not cause fragmentation. Finally, the varying compressibility of workloads means that the cache may be limited by either tag storage or data storage, with each storage structure requiring a separate and different replacement policy.

### 3.2.1 Cache Architecture

2DCC cache consists of three structures: Data array, Tag array and Hash array. Figure 3.3 illustrates these three structures and shows how these components are interconnected with pointers.

#### Data Array

Storage of variable-sized blocks is accomplished by segmenting each set in the data array into eight-byte segments (similar to prior work [200]): a single cache block may occupy from one up to eight contiguous segments depending on the compression factor.

Because the tag array is decoupled from the data array (unlike in [200]) and the cache can store more tags than uncompressed blocks, 2DCC may

need to evict blocks when space in the data array runs out even if some tags are still free. To identify the tags that point to a given data segment, 2DCC uses a per-segment back-pointer to one of the corresponding tags.

To support the data array replacement policy, each segment also stores a count of tags pointed to it. A free-list bit vector is used to allocate entries and manage free space in the data array. Upon insertion in the cache, the controller consults this free-list to find an empty set with free blocks, and the compressed is inserted there. This helps the replacement policy to avoid evicting blocks due to space limitation in a particular set, while there are empty sets (example in section 3.2).

### Tag Array

As in a conventional set-associative tag array, each entry contains the tag itself, tag replacement policy state, and validity/coherence state. Each entry also specifies the compression encoding. The tag entry also contains a "data pointer" to identify the segment(s) storing the cached block.

Finally, multiple tags that point to the same data segment form a doubly-linked list, used to remove all tags associated with an evicted block, and to form a free-list of unused tags.

### Hash Array

To detect identical cache blocks, 2DCC needs to compare the *contents* of an incoming block with the contents of blocks that are already cached. Naturally, scanning through the entire cache is not an option. Instead, 2DCC detects candidates for deduplication by storing hashes of block contents in a separate small hash array. Because the common case is that incoming lines are unique, the hash array essentially serves to filter out most lines that cannot be deduplicated.

The hash array is a set-associative table. Each entry points to the data array segment where the original block is stored. (This is safe even if the original block is modified or evicted, as hash collisions mean that hash matches must in any case be verified against the full cache block.) Based on our experiments, storing only 1024 hashes is the hash array is sufficient to capture nearly all possible deduplication while reducing the hash collisions to less than 1%.

In operation, each incoming block's contents are hashed using a 64-bit $H_3$ hash [214]. If the hash is not found, insertion proceeds normally. If the hash matches (i.e., a duplicate block may already exist in the cache), the

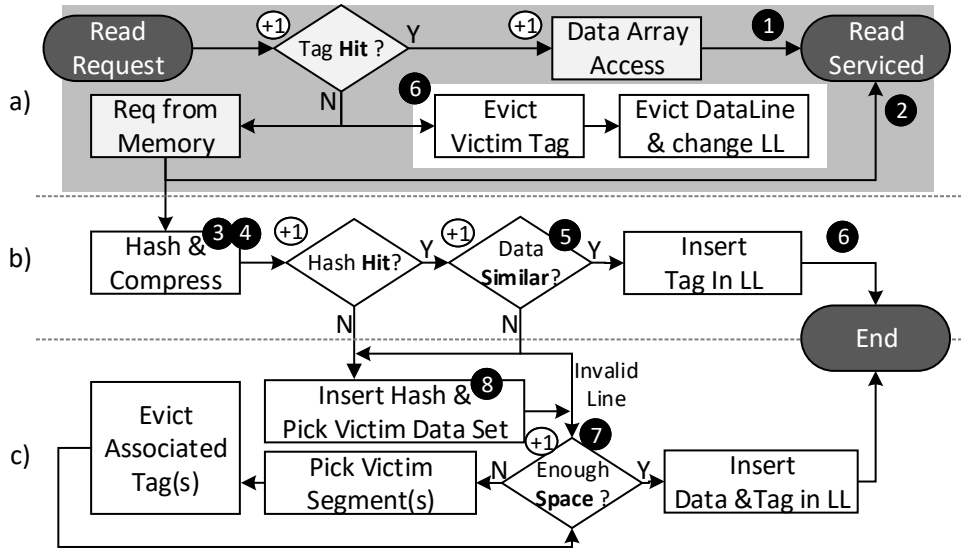



Figure 3.4: Read access flowchart in 2DCC. Shaded (a) = on the critical path; unshaded (b, c) = off the critical path. LL = linked list of deduplicated tags.

block it points to is fetched from the data array and the actual data are compared; if the data are identical, then the line is deduplicated, otherwise it is inserted as a new block.

### 3.2.2 Cache Operation

2DCC operation is largely similar to that of a conventional cache, with some differences due to tag/data array decoupling and the need to deduplicate and compress stored data. We detail those differences below.

*Reads, evictions, and insertions.* The operation of these accesses is illustrated in Figure 3.4. The critical-path portion of read accesses — both hit ❶ and miss ❷ — corresponds to a conventional cache.

The hash and the compressed line size are calculated off the critical path ❸ ❹. If the hash exists in the hash array, the new block is a deduplication candidate, and the existing block is retrieved from the data array and compared against the newly arrived data ❺ to determine if the block is a duplicate.

If the entry is to be deduplicated, an unused tag is obtained either from the tag free list or by evicting an existing tag ❻. If the entry cannot be deduplicated, a data entry is also allocated, possibly following an eviction

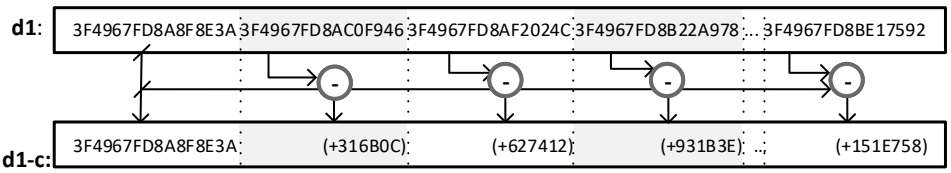



Figure 3.5: Example of compressing a 64-byte cache block from roms_r (*d1* from Figure 3.6). The block consists of 64-bit floating-point numbers whose values are close; they are compressed to a 64-bit base value followed by eight 32-bit offsets, for a total compressed size of 36 bytes.

of some data segments and their associated tags ❼. If the entry was not deduplicated, its hash is also inserted into the hash array to enable future deduplicaton ❽.

*Writes.* Writes reflect those in a conventional cache: with inclusive write-back caches, which we use in this work, write requests always hit, and execute off the critical path.

Writes may also change the compressed size. In parallel to the tag access, therefore, the hash of the contents is computed; if this hits in the hash array, the relevant block is fetched and compared to the newly written data. If the newly written block can be deduplicated, the data pointer swings to the existing copy and the redundant segments are freed.

If the written block cannot be deduplicated, it is first re-compressed. If it fits in the same number of segments, the data array entry is overwritten, possibly freeing some segments. If the line is larger, victim segments are selected from the data array before inserting the block as if it were a new insertion.

*Intra-block compression/decompression.* For compressing individual cache blocks, we use the BΔI compression method [200]. Briefly, BΔI calculates the mean of the words in the block to determine the number of bytes needed to express the distance from this base value or from 0. If all distances can be expressed in fewer bytes than the original value (e.g., 4 bytes), the compressed block consists of the base value followed by a sequence of distance offsets used to reconstruct the original words in the cache block. Decompression consists of adding the offsets to the base, and is completed in one cycle.

Figure 3.5 shows an example of this process. The block (*d1* from Figure 3.6) consists of 64-bit floating-point numbers whose values are close. The intra-block compression reduces the block to 40 bytes (an 8-byte base value

followed by eight 4-byte offsets), and compacts it to take up 5 segments in the set.

### 3.2.3 Walk-through Example

In order to better understand the operations, we describe the cache operation by tracing "the life of a cache block" on a tiny version of 2DCC in Figure 3.6 with an example from the roms_r ocean simulation workload. We begin with the state in panel (a), with one uncompressible, unduplicated block in the cache with tag t0 and data d0, such as block L0 in Figure 3.2(a).

In panel (b), a lower-level cache requests an address with tag *t1*, which misses in the tag array ❶ and triggers a backing memory request for its data *d1*. When *d1* arrives, it is compressed to *d1-c*, and, in parallel, hashed to search for duplicates ❷. The hash is then looked up in the hash array to determine whether the block can be deduplicated, but as it is the first occurrence of this data, the lookup fails.

To insert the new block in the cache, the controller consults the freelist to find that set 1 has a free block, and the compressed *d1-c* is inserted there ❸. At the same time, *t1* is inserted in the tag array with its data pointer set to point at *d1-c* and vice versa ❹, and the hash for *d1* is inserted in the hash array ❺.

In panel (c), a lower-level cache requests an address with tag *t2*, which also misses in the tag array ❻; this triggers another memory request. Once data block *d2* arrives, it is compressed (to *d2-c*) and hashed as before ❼. This time, however, the hash hits in the hash array with the entry pointing to *d1-c*, indicating that *d2* is a possible duplicate of *d1*; to verify this, *d1-c* is retrieved and compared against *d2-c*. An exact match is determined, the deduplication count in *d1-c* is incremented ❽, and *t2* is inserted into the tag array pointing to *d1-c* ❾.

### 3.2.4 Replacement Policies

As described in earlier, 2DCC has three decoupled structures: a tag array, a data array, and a hash array. Unlike in a conventional cache, the three arrays have different goals and need different replacement policies.

*Tag array:* The goal of the tag array replacement policy (RP) is to preferentially retain addresses likely to be accessed in the future. The RP should therefore be the same as the equivalent conventional cache RP. In this work, we use the least-recently-used victim selection with most-recently used insertion (LRU), but other eviction policies may be more appropriate for large

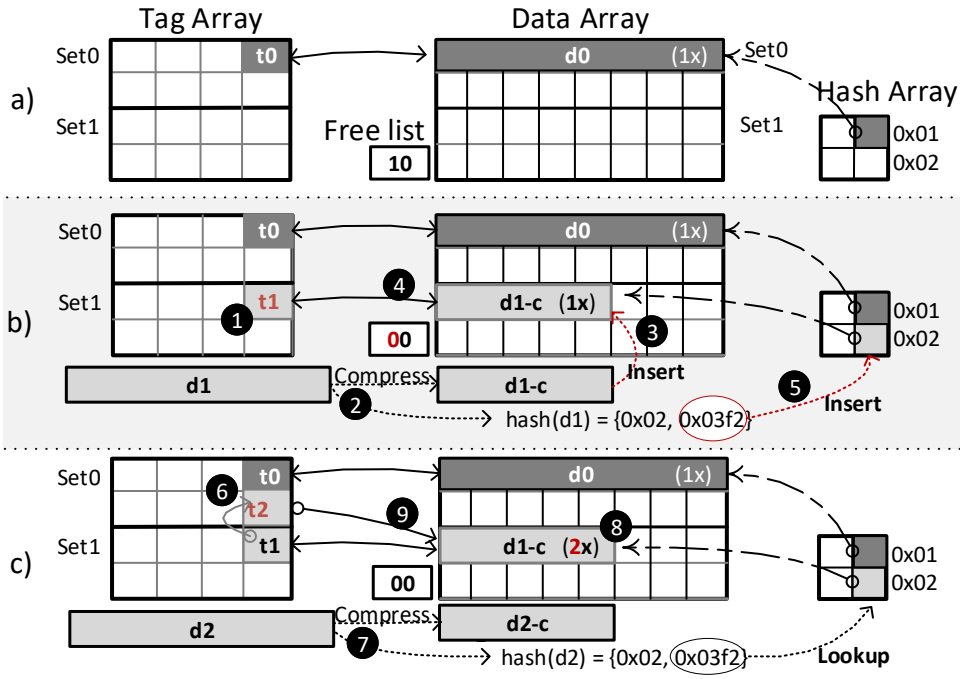



Figure 3.6: Example of 2DCC operation on lines from roms_r benchmark: (a) initial state; (b) insertion of unique but compressible block; (c) insertion of a new block whose data is identical to that of panel (b).

caches and specific workloads [86, 252, etc.].

*Data array:* The data array, on the other hand, provides *storage space* for the blocks identified as likely to be re-referenced. The storage is many-to-one: when several cache blocks contain the same data, one data array entry will be shared among several tags. When evicting a data array entry, all of the tag array entries that point to it must also be evicted. This makes conventional cache victim policies, which do not account for the cost of evicting multiple tags, unsuitable.

Observe that the policy does *not* need to consider which blocks are likely to be re-referenced, as the tag array replacement policy already ensures that only useful blocks are cached. The goal of the data array, therefore, should be to *enable the tag array to store more blocks*. Our policy has three stages:

1. If a set in the data array is free, insert the block there.
2. Otherwise, attempt to find space in a partly occupied block: randomly select four sets, and, if one of them has enough space, insert the new block there.
3. Finally, examine the four blocks from step 2, and select the one that (a) has enough space, and (b) minimizes the number of evicted tags from the tag array.

In effect, this process combines a random sampling process with a selection policy that retains the most deduplicated entries.

*Hash array:* The main purpose here is to enable deduplication of blocks stored in the data array. Thus, the hash array should identify (a) currently cached blocks whose contents are likely to reappear in other, soon-to-be-accessed blocks, and (b) incoming blocks whose contents are likely to reappear later. We therefore use the LRU policy applied to content hashes.

## 3.3 Methodology

We extended ZSim [213] to implement 2DCC and the state-of-the-art hardware compression techniques for intra-block compression (B$\Delta$I [200]) and inter-block deduplication (Dedup [242]). We modeled detailed event timing and interconnect congestion for both on- and off-critical-path events. The simulated system is shown in Table 3.1; compression was applied to the L3 level only. We used CACTI 6.0 [138] to estimate silicon area requirements, including all data structures for each compression method. For all performance studies, we normalized the three designs to the same silicon area.

| | |
|---|---|
| CPU | i5-750-like: x86-64, 2.6GHz, 4-wide OoO, 80 entry ROB |
| L1I | 32KB, 4-way, 3 Cycle access lat, 64B lines, LRU |
| L1D | 32KB, 8-way, 4 Cycle access lat, 64B lines, LRU |
| L2 | Private, 256KB, 8 way, 11 Cycles lat, 64B lines, LRU |
| L3 | Shared 1 MB, 8-way, 39 Cycles lat, 64B lines, 8 banks |
| Memory | DDR3-1066, 1GB |

Table 3.1: Configuration of the simulated system.

We used an extensive set of integer and floating-point applications from SPEC CPU2017 [14] and PARSEC [38], as well as those applications from SPEC CPU2006 [114] that are not in CPU2017. All were run with large input sizes (native in Parsec and reference in SPEC). Simulations skipped the first 40 billion instructions, and then sampled the last 20% of each 1 billion instructions for a total of 40 billion instructions.

**Sizing data structures**
Sizing decoupled structures (tag, data, and hash arrays) under a fixed silicon area budget is key to our design. In 2DCC, we must make two sizing decisions: (a) the ratio of tags to raw data blocks (which must exceed 1 to enable compression) and (b) the size of the hash array that captures inter-block redundancy.

**Tag array vs. data array**
We observed that the compressibility of cache blocks varies not only among applications, but also among different phases within an application, from as low as 1× in *streamcluster* to more than 10× in *fotonik3d_r*. Similar to prior work [242], we allow the cache to store four times more compressed lines than uncompressed lines.

**Hash Table**
For the hash array, the tradeoff is between, on the one hand, reducing the silicon footprint to make more space for tags and data entries and, on the other hand, making it large enough to capture enough of the inter-block redundancy.

To examine the design space, we compared an oracle hash table — which searches the entire cache for a match — against hash array sizes from 64 to 16,384 entries. In our experiments, 99.2% of the locality was captured with

| | | Baseline | BΔI [200] | Dedup [242] | 2DCC |
|---|---|---|---|---|---|
| **Tag** | #Entries | 16384 | 49152 | 40960 | 36864 |
| | Entry Size | 39b | 49b | 85b | 93b |
| | Total Size | 78KB | 294KB | 425KB | 414KB |
| **Data** | #Entries | 16384 | 12288 | 10240 | 9216 |
| | Entry Size | 512b | 512+0b | 512+16b | 512+104b |
| | Freelist | | | | 1152b |
| | Total Size | 1024KB | 768KB | 660KB | 694KB |
| **Hash** | # Entries | - | - | 1024 | 1024 |
| | Entry Size | - | - | 3B | 3.375B |
| | Total Size | - | - | 3KB | 3.375KB |
| | **Total Size** | **1.08MB** | **1.05MB** | **1.07MB** | **1.08MB** |

Table 3.2: Storage allocation. All compressed caches are sized to fit in the same silicon size of a 1MB conventional cache with 48-bit address space.

1,024 entries (64 sets × 16 ways), with a collision rate of $< 1\%$. We use this hash size for the remainder of the experiments.

**Silicon area allocation**

We configured 2DCC as well as our three baselines to match that of a conventional, uncompressed cache with 1MB of data storage. For the compressed caches, the total space available in the data array is less that 1MB because more of the available silicon budget must be dedicated to tags; Table 3.2 shows space allocation details.

Metadata for each conventional cache tag entry consists of valid, dirty, and LRU bits. 2DCC adds 4 bits for the compression type encoding, 32 bits for the previous and next tag pointers, and 17 bits for the data pointer (11 bits to index the set and 6 bits to index the segment within that set). BΔI adds 10 bits over the conventional cache for compression-type encoding and the segment pointer. Finally, Dedup tag entries add 32 bits for tags pointers and 14 bits for the data pointer. Data array overheads in 2DCC are 16 bits per segment for the tag list pointer, while Dedup has one 16-bit pointer for each cache block. 2DCC also requires a hash array, with each entry consisting of 10 bits for the hash tag and 17 bits for the data segment pointer; Dedup has a similar table but uses only 14 bits for the data pointer.

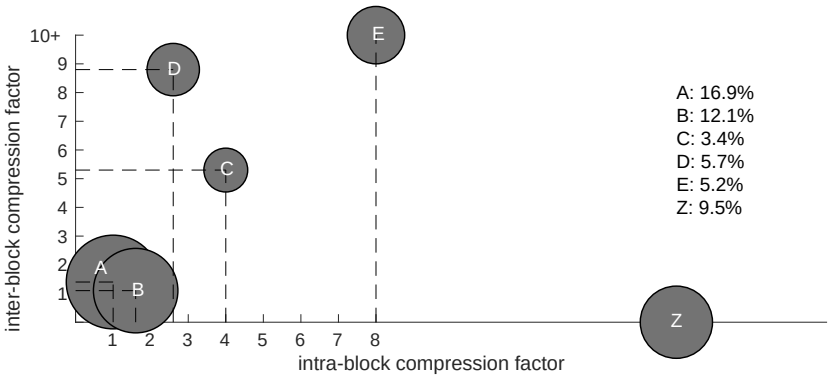



Figure 3.7: The bubble sizes represent storage savings due to combined intra- and inter-block compression, plotted against different compression factors. Z is the amount of savings due to all-zero blocks.

## 3.4 Evaluation Results

### 3.4.1 Effectiveness of 2D compression.

Figure 3.7 shows the inter-block compression factor for each possible intra-block compression factor averaged over all benchmarks; the bubble size shows how much cache area was saved due to a specific combination. The largest savings — 16.9% of cache — come from blocks that cannot be compressed by themselves, but can be deduplicated, on average, 1.4× (bubble A). The next 12.1% is saved by blocks that cannot be deduplicated, but are amenable to intra-block compression with a compression factor of 1.6× on average (B). Significant additional savings (14.3% cache space total) come from blocks that are both compressible within each block but also identical to other blocks (C, D, E). Finally, 9.5% cache space is saved by using a special representation for zero-only blocks.

This validates the 2DCC intuition: both choosing the appropriate compression for each block (A, B) and using both compress

### 3.4.2 Compression Analysis

Figure 3.8(a) shows the cache space needed by different workloads using 2DCC compared to state-of-the-art methods for intra- (B$\Delta$I [200]) and inter-block compression (Dedup [242]), normalized to a conventional cache. The compressed size indicates the ratio of number of tags in the cache over the number of data blocks (compressed and uncompressed) which varies among

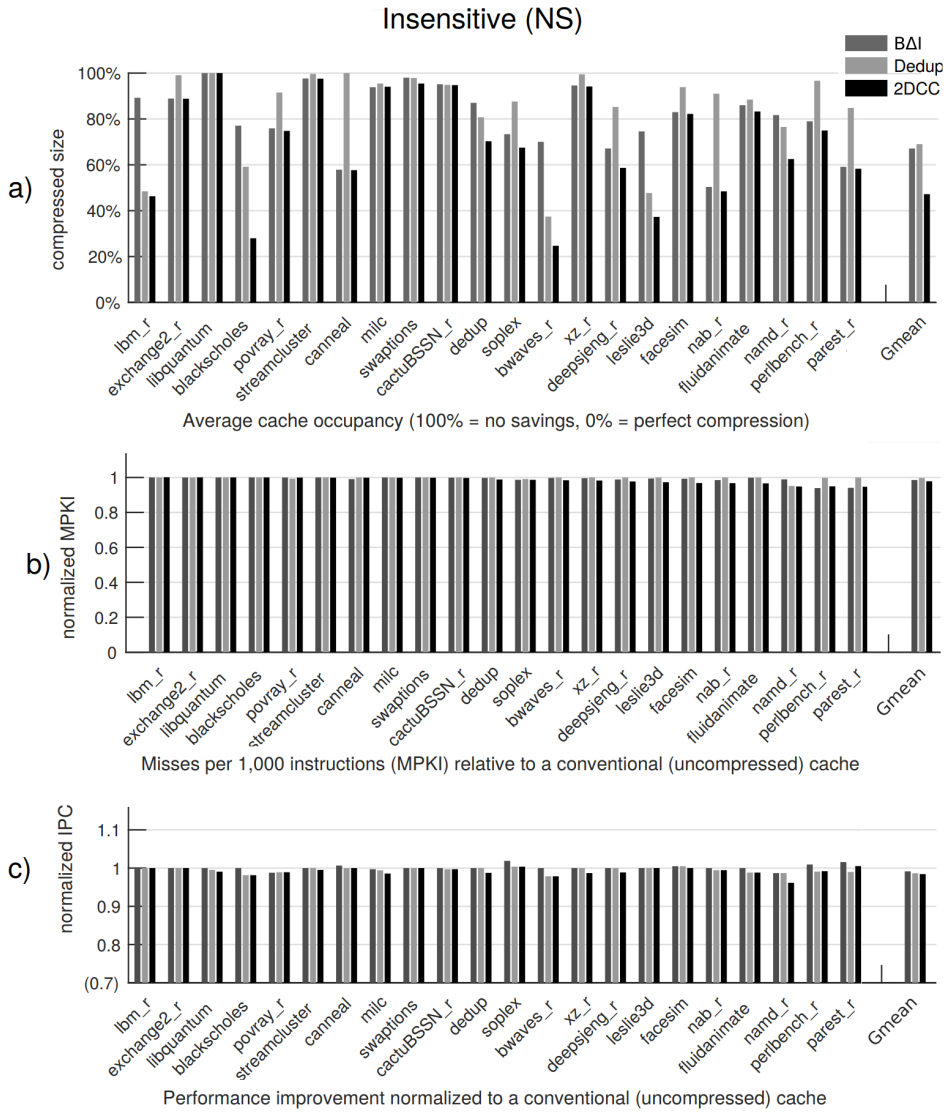



Figure 3.8: Cache occupancy, cache miss rates, and performance improvements of 2DCC compared to iso-silicon BΔI (intra-block) and Dedup (inter-block) caches, normalized to an iso-silicon conventional (uncompressed) cache for the cache-insensitive subset of benchmarks.

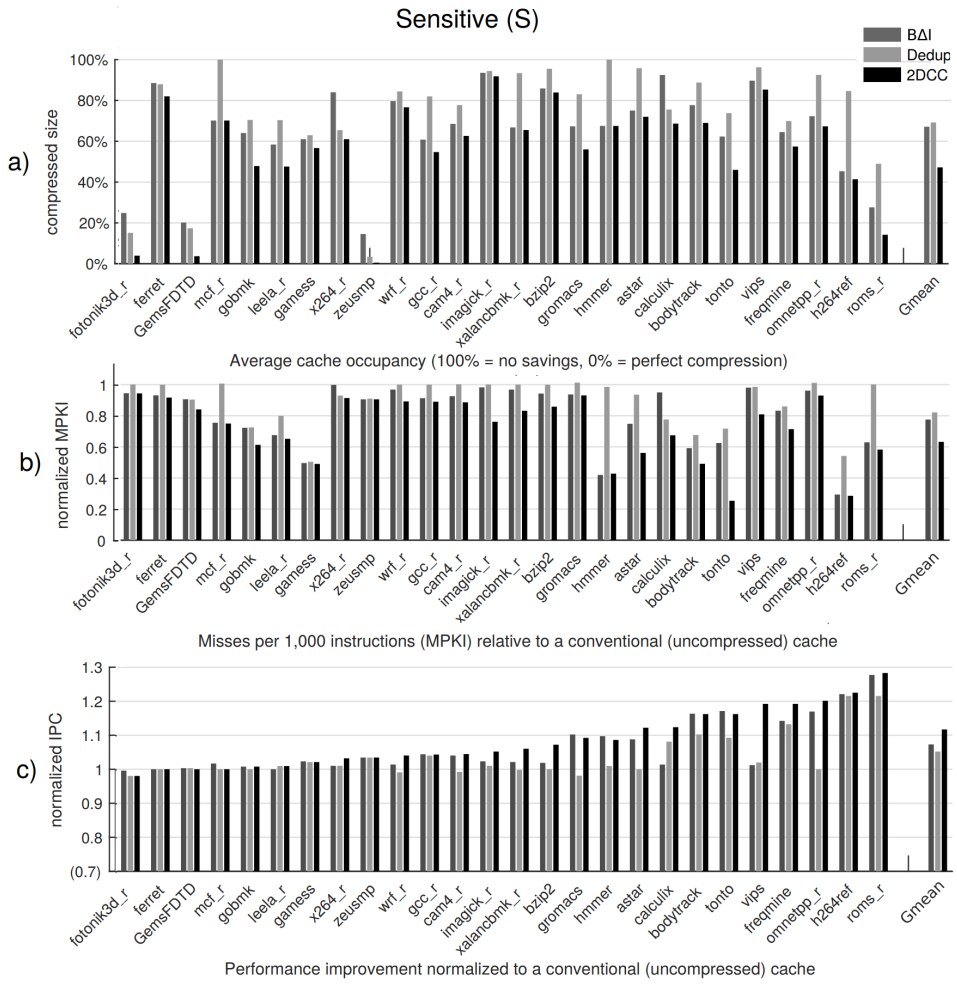



Figure 3.9: Cache occupancy, cache miss rates, and performance improvements of 2DCC compared to iso-silicon BΔI (intra-block) and Dedup (inter-block) caches, normalized to an iso-silicon conventional (uncompressed) cache for the cache-sensitive subset of the benchmarks.

different phases within an application. We report averages over the entire program runtime from an execution-driven simulation (see Section 3.3). All caches take the same silicon area as a 1MB conventional cache (see Table 3.2).

On average, 2DCC is able to reduce the cache footprint to 47.2% of the original footprint (i.e., 2.1× compression), a substantial improvement over BΔI (67.1%) and Dedup (69.2%).

### 3.4.3 Miss Rates Analysis

We divided the benchmarks into cache sensitive (S) and cache insensitive (NS): we consider a benchmark to be cache insensitive if there is $< 3\%$ change in MPKI when the conventional LLC size is doubled (this typically means that their workloads mostly fit in the L2 or even L1D cache).

Figure 3.8(b) shows that 2DCC reduces cache misses per 1,000 instructions (MPKI) by 1.6× compared to 1.3× for BΔI and 1.2× for Dedup on average for the cache sensitive benchmarks. At the same time, the MPKI impact of cache compression on the cache-insensitive benchmarks is negligible (1.6%)

### 3.4.4 Cost Analysis

As with every compressed cache, there can be various overheads:

**Area Impact**: As described in section 3.3, all compressed caches are sized to fit in the same silicon size of a 1MB conventional cache. Therefore, 2DCC incurs no area overheads.

We also implemented and synthesized the BΔI scheme [200]. The compression/decompression units require $0.037\text{mm}^2$ (20k NAND gates) for its logics, which at 45nm is less than 1% of the silicon area required for even a 1MB cache.

**Latency Impact**: Latency overheads of 2DCC is very similar to Dedup cache; as our technique and Dedup breaks the direct mapping between the tag and data array, this means one additional data array lookup for locating a data block in data array. This is a sequential event and cannot be masked by requesting the tag and data in parallel, therefore we model it by adding an extra access latency.

Another additional latency is due to having the compressed blocks. To model the additional latency due to decompression, which is also on the critical path of read requests, we use +1 cycles for both BΔIand 2DCC which is on par with [200]. We also experimented with +2 and +3 cycles

| Cache | Size(MB) | Dynamic read energy | Leakage power |
|---|---|---|---|
| Conv. | 1 | 0.35 nJ | 677.66 mW |
| BΔI | 1 | 0.37 nJ | 679.21 mW |
| Dedup | 1 | 0.39 nJ | 699.70 mW |
| 2DCC | 1 | 0.39 nJ | 695.16 mW |

Table 3.3: Dynamic energy and leakage power of compressed caches and conventional of 1MB (silicon area of $2.52mm^2$) in $32nm$ technology.

and observed that when decompression latency of 2DCC increases from 1 to 3 cycles, performance degrades by less than 0.6% on average.

**Multiple tag eviction**: In a compressed cache, there are cases which multiple cache blocks may need to be evicted because evicting a single cache block may not create enough space for the incoming or modified block. First, when a new cache block is inserted into the cache. Second, when a block already in the cache is modified such that its new size is larger than its old size. In both cases, with the help of replacement policy, the cache evicts multiple cache blocks to create enough space for the incoming or modified blocks. Such a policy can increase the latency of eviction due to multiple tag evictions. We also investigated whether evictions of multiple tags are a significant problem, by measuring the ratio of evicted tags to cache accesses. Because of its better compression, 2DCC has the lowest eviction rate of 0.032 evictions per access, compared to 0.049, 0.042, and 0.041 for the conventional cache, BΔI, and Dedup, respectively. This means that multi-tag evictions are very rare, and do not have any performance impact.

**Energy and power Impact**: We used CACTI [138] to measure the latency, read energy, and leakage power of 2DCC and the three baselines (see Table 3.3); results show that 2DCC uses 11% more energy for each read, and has a 2.5% leakage power overhead. The added per access overhead of 0.04nj is trivial compared to the energy of accessing external DRAM (32.61nJ using the same CACTI model). This means that 2DCC can actually save energy when the entire memory hierarchy is considered (by calculating the total added power of compressed cache and total power saved by accessing off-chip DRAM less frequently). We investigate this in more details in the following chapter.

The BΔI decompression/decompression power is also 7.4 mW/20.59 mW on average [200] which is a negligible number compared with the power consumption of the entire processor.

### 3.4.5 Speedup Analysis

Figure 3.8(c) shows that the lower MPKI allows 2DCC to improve performance (IPC) by 11.7% for the cache-sensitive benchmarks, vs. 7.3% for B$\Delta$I and 5.2% for Dedup. Cache-insensitive benchmarks can suffer a slight performance degradation (avg. 2.6%): for example, *bwaves* is highly compressible but cache-insensitive, so the compression/decompression latencies are not offset by more frequent cache hits.

## 3.5 Reducing Redundancy with Approximation

### 3.5.1 Approximate Value Locality In Caches

Prior approximate cache proposals [173, 211] have observed that many values stored in a cache are so close that replacing one value by another makes little difference to the effectiveness of many applications; approximate caches can substantially increase effective cache capacity by taking advantage of the *approximate value locality*. This locality can be converted to increased cache capacity by detecting cachelines with approximately equal value sequences, storing only one of those lines in the cache, and returning an acceptable approximation when a line is retrieved. For example, Doppelgänger [173] treats the average of all values in the cacheline as a "signature" and stores only one representative cacheline for each signature.

### 3.5.2 Decoupling Compression and Approximation

Existing designs, share two significant limitations. First, because they are in effect lossy compression techniques, they can only compress data that identified as approximable (e.g., by the programmer), and are of little use for applications where approximation is not practical. Second, because approximation is an integral part of the lossy compression mechanism, they can only compress approximable data that fit a single redundancy pattern: for example, Doppelgänger captures value similarity *between* cachelines, but ignores value similarity *within* each cacheline [173], and Bunker Cache is only effective on image-like data [211]. Both of these are serious barriers to adoption in commercial CPUs.

We argue that approximation and compression are orthogonal, complementary techniques that should be *decoupled* in cache designs. Such a design, illustrated in Figure 3.10, would comprise two stages: (a) a lightweight approximation stage, applied only to data identified as approximable, and (b) a cache compression stage, applied to all data, that leverages one of the

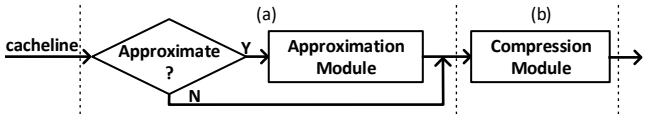



Figure 3.10: A decoupled approximate cache design that separates the approximation (a) and compression (b) aspects.

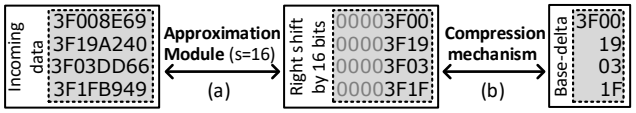



Figure 3.11: Approximating and compressing a cacheline from *jmeint* [178, 263] using a base-delta representation [200].

many existing cache compression proposals [21, 26, 56, 88, 191, 200, 242**?** ]. To demonstrate the practicality of this approach, we combine approximation with 2DCC as well as the two other representative compressed caches, B$\Delta$I [200] and Dedup [242]. The decoupled paradigm allows the designer to choose a compression algorithm that is suitable for the application rather than one that is dictated by the approximate cache design, which in turn can result in better space savings: simulations on a range of applications from AxBench [263], Parsec [37], and SPEC [114] show that, under the same quality-of-results criteria, decoupled designs achieve up to 63% more compression (7.4× geomean) than a bespoke design like Doppelgänger.

Importantly, decoupled designs can also compress non-approximable data, potentially making approximate caches more attractive for commercial applications.

### 3.5.3 Methodology

**Quality of results criteria**

For all benchmarks, we determined quality cutoffs where degradation was (a) barely noticeable visually or (b) did not compromise the purpose of the benchmark. The quality metrics are shown in Table 3.4.

We focused on floating-point array structures, and applied the maximum single approximation level suitable for the entire workload; more approximation may be possible by using per-datastructure levels. Approximation and compression are applied only at the LLC level whenever a new cacheline is brought in from DRAM; decompression and deapproximation are applied

| BM | fp64/fp32 | Quality | BM | fp64/fp32 | Quality |
|---|---|---|---|---|---|
| swaptions | 45/0 | MSE<1 | fft | 0/17 | MSE<82.5 |
| streamcluster | 0/11 | AE=0% | sobel | 0/18 | PSNR>60 |
| calculix | 47/0 | MSE=0 | vips | 43/17 | PSNR>60 |
| inversek2j | 0/17 | MSE<5e-3 | milc | 32/0 | MSE<5e-8 |
| blackscholes | 0/18 | MSE<5e-3 | ferret | 0/5 | AE=0% |
| K-Means | 0/6 | PSNR>50 | namd | 32/0 | AE<1e3 |
| cactusADM | 38/0 | MSE<2e-7 | jmeint | 0/17 | RE<5e-3 |

Table 3.4: Quality cut-off levels for fp64/fp32 and QoR criteria.

whenever the line is read or written.

**Bespoke approximate cache**
We implemented Doppelgänger [173], the best-performing approximate cache proposal to date (the more recent Bunker Cache [211] has worse compression [211, fig. 22]) Value ranges for the approximation maps were taken from recorded runtime minimum and maximum values in the manually annotated approximable regions of each benchmark using the 14-bit map space.

**Decoupled approximation scheme**
To approximate floating-point numbers, we zero the $s$ rightmost bits of the mantissa, where $s$ is chosen per application according to the QoR metrics above; we then right-shift the entire value by $s$ bits. When an approximate value is fetched, we left-shift the stored value by $s$ bits and use that to service the request. This mechanism, illustrated in Figure 3.11(a), naturally lends itself to a simple, fast hardware implementation.

Like prior work [173, 211, 212], we assume that approximate data structures are annotated by the programmer. In our design, the approximation level $s$ is included in every cache request, as is a single bit identifying the datatype (fp32 or fp64); this can be implemented either by extending the ISA with approximate variants of load/store instructions or by adding a hardware region lookup table that can be filled by the application and consulted during load/store execution.

To fairly compare with Doppelgänger, we only approximate to cachelines where all elements have the same type.

**Decoupled compression schemes**
Compression is applied after approximation and before deapproximation, as shown in Figure 3.11(b). We combined this approximation scheme with three

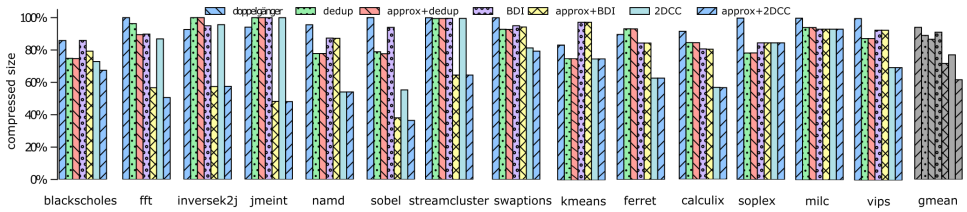



Figure 3.12: Compressed working sizes for Doppelgänger and several decoupled approximation+compression combinations.

representative cache compression techniques: base-delta-immediate compression (B$\Delta$I) [200], exact deduplication (Dedup) [242], and our proposal, 2DCC.

### 3.5.4 Evaluation Results

Figure 3.5.2 shows the savings from compression and approximation normalized to the uncompressed footprint for all caches, measured as a running average over each workload's region of interest. The decoupled approximate caches reduce workload footprints to as little as 36.5% (sobel) and 60% on average (gmean); all outperform Doppelgänger, which only manages to reduce the footprint to 95% (gmean).

This is largely due to two factors. One is that the decoupled approximate caches capture the intra-line redundancy created by the approximation stage (e.g., fft, inversek2j, jmeint, sobel, streamcluster); because Doppelgänger effectively implements near-deduplication, it cannot capture intra-line effects. The other is that the decoupled designs can also compress non-approximable data (e.g., ferret).

Overall, these results make a case for decoupling approximation and cache compression — general-purpose designs where approximation is implemented as a separate, optional stage in the cache compression pipeline. These achieve better compression than bespoke approximate caches, and offer a practical strategy for introducing approximation in commercial cache hierarchies.

## 3.6 Summary

This chapter, demonstrates that a simple technique that takes advantage of both types of of redundancy results in improvement in compression for several applications: some workloads exhibit either inter-block or intra-block

redundancy, while others exhibit both. It proposes 2DCC, a simple technique that fills this gap.

This chapter also demonstrates that approximation and compression are orthogonal, complementary techniques that should be decoupled in cache designs. Compressed caches can be comprise a lightweight approximation stage and an effective cache compression stage.

# Chapter 4

# Reducing Data Redundancy via Dynamic Clustering

In this chapter[2], we leverage similarity across memory data blocks in order to reduce the redundancy in cache memories.

Below, we first describe the missed opportunity in compressing multiple cachelines if we only consider exact values (duplicates). We demonstrate significant *similarity but not identity* (i.e., near-duplicates) in the data values of memory blocks *across* different cachelines for a broad range of workloads. With that intuition in mind, we describe the opportunities for in-cache clustering of the data. Then, we propose Thesaurus, a dynamic inter-cacheline compression technique which uses dynamic clustering to efficiently detect and compress groups of similar memory blocks. Thesaurus uses a novel, hardware-friendly locality-sensitive hashing design in order for the approximate near data search. It continues the chapter with the architectural and operational changes needed for this proposal as well as the detailed description on the compression formats. We also develop a replacement policy for the data array in the LLC that balances the development of new clusters with conserving existing clusters. Finally, we evaluate this design on an extensive set of benchmarks.

## 4.1 Near-Exact Data Redundancy

As described in chapter 2, several existing compression schemes [20, 21, 25, 26, 56, 76, 187, 191, 200, 218–220] take advantage of low entropy of data within a small memory block (e.g., a cacheline) by compressing each block independently. On the other hand, state-of-the-art inter-cacheline compression schemes such as exact deduplication [242], work on larger parts of the memory (e.g., multiple memory blocks) and can reveal redundancies only when multiple cachelines are considered.

[2]Parts of this chapter appear as: A. Ghasemazar, P. Nair, M. Lis, "Thesaurus: Efficient Cache Compression via Dynamic Clustering," In ASPLOS, 2020.

Although exact deduplication can capture structural properties of more substantial, heterogeneous data structures, it can only exploit data regularity if multiple LLC lines have *exact* data values [242]. To see how much opportunity is lost, consider an ideal inter-cacheline compression scheme that inserts data by searching the entire LLC for similar cachelines and stores only the bytes that differ from the most similar existing cacheline whenever this representation is smaller than an uncompressed cacheline; we refer to this setup as *Ideal-Diff*. Figure 4.1 shows the effective LLC capacity for (a) a system without compression, (b) an idealized deduplication scheme that also instantly searches the LLC for exact matches (*Ideal-Dedup*), and (c) *Ideal-Diff*, on SPEC CPU 2017 suite [43]. The potential of detecting and compressing similar lines is significant: *Ideal-Diff* increases the LLC capacity by 2.5× over the baseline (geomean), compared to only 1.3× for *Ideal-Dedup*.

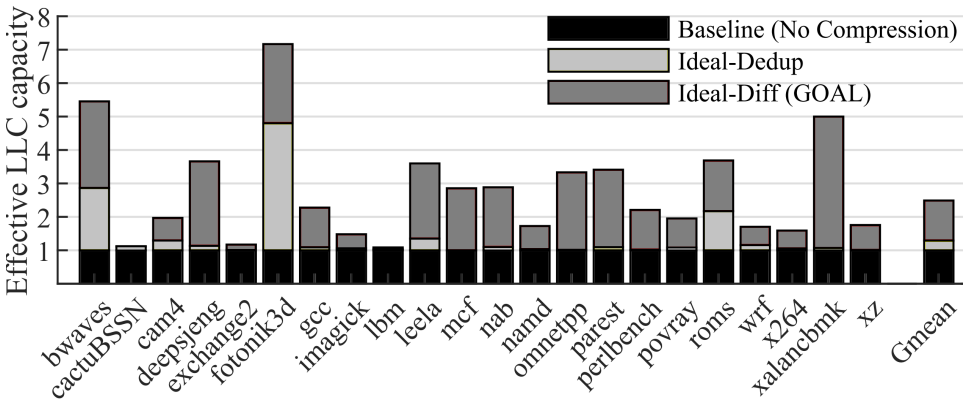
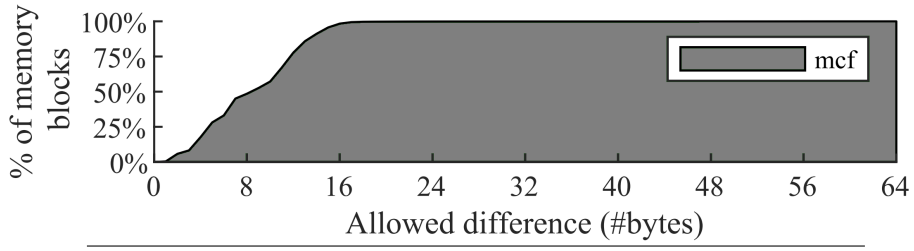



Figure 4.1: Effective LLC capacity from data compression by executing the SPEC-2017 suite [43]. On average, Ideal Deduplication (*Ideal-Dedup*) improves the effective LLC capacity by 1.3×. However, *Ideal-Diff*, that groups nearly identical memory blocks, can increase effective LLC capacity by 2.5×.

To achieve good compression with *Ideal-Diff*, the overheads of storing diffs must be low (i.e., the diffs must be relatively small). We observed that this tends to be true for a wide range of workloads. For example, Figure 4.2(top) illustrates this using an LLC snapshot of the *mcf* workload from SPEC CPU 2017 [43]. The working set contains very few duplicate memory blocks, making exact deduplication ineffective. Intra-cacheline techniques also have limited effectiveness, as the primary datatype contains a variety




$l_1$: 00002AAAC02419D8 00002AAAC0237610 0000000000000000 FFFFFFFFFECEF790 00006A04FFFFFE67 ⋯
$l_2$: 00002AAAC024B8B0 00002AAAC023EE60 0000000000000000 FFFFFFFFFECEF73C 00006A2CFFFFEEE7 ⋯
$l_3$: FFFFFFFFFFFFFFC 4 00002AAAC11FE988 0000000000000000 0000000001312724 FFFFFFFFFECED8DC ⋯
$l_4$: FFFFFFFFFFFFFFC 4 00002AAAC11FA920 0000000000000000 00000000013123A0 FFFFFFFFFECEDC60 ⋯

Figure 4.2: Top: Fraction of 64-byte cachelines in an LLC snapshot of *mcf* that can be deduplicated with at least one other cacheline *if* differences up to $n$ bytes are permitted for $0 \leqslant n \leqslant 64$. Bottom: Two clusters of near-duplicate cachelines from *mcf*.

of fields with different types and ranges (see Listing 4.1).

On the other hand, exploiting similarity across cacheline boundaries and relaxing the exact-duplicate requirement is very effective: almost *half* of the cached memory blocks differ from another block by a maximum of 8 bytes, and nearly *all* memory blocks differ only by a maximum of 16 bytes. Therefore, we can obtain significant LLC storage capacity by storing 16-byte diffs instead of full 64-byte blocks.

## 4.2 Capturing Near Exact Data

Unfortunately, working sets usually do not contain a single reference memory block around which all other memory blocks could cluster; on the contrary, we may require *several* reference memory blocks with vastly different data values. This is the case in *mcf*: in Figure 4.2(bottom), lines $l_1 \ldots l_4$ all come from the same `node` data structure in *mcf*, but only $\{l_1, l_2\}$ and $\{l_3, l_4\}$ are near-duplicate pairs. This is because `node` takes up 68 bytes (see Listing 4.1) and is not aligned to the 64-byte cacheline size. The misalignment naturally creates several "clusters," each with its own reference memory block referred to as the "clusteroid." To achieve effective compression, therefore, multiple clusters must be identified; in addition, because the contents are input-dependent, this must happen dynamically at runtime.

```
struct node {
    val (8 Bytes) potential;
    val (4 Bytes) orientation;
    ptr (8 Bytes) child, pred
    ptr (8 Bytes) sibling, sibling_prev;
    ptr (8 Bytes) basic_arc,firstout;
    ptr (8 Bytes) firstin, arc_tmp;
    val (8 Bytes) flow;
    val (8 Bytes) depth;
    val (4 Bytes) number;
    val (4 Bytes) time;
};
```

Listing 4.1: The `node` data structure in *mcf*

## 4.3 The Opportunity for In-Cache Clustering

To determine whether in-cache clustering is practical, and whether it should be dynamic, we asked three questions:

1. Do caches contain clusters with enough elements to provide substantial opportunities for compression?
2. Are there few clusters with clusteroids that could be hardcoded, or must clusteroids be computed at runtime?
3. Do cluster count and size vary among workloads enough to need a runtime-adaptive solution?

To answer these questions, we performed DBScan clustering [80] on LLC snapshots from the SPEC CPU 2017 suite, configuring similarity criteria for each workload to target 40% space savings. The experimental setup here is idealized in two ways: (a) the algorithm sees the entire LLC at once rather than each memory block separately at insertion time, and (b) DBScan uses far more computation and storage than is practical to implement within a cache controller.

Figure 4.3 shows that the LLC in most workloads has significant clusters of 10 or more members, with many exhibiting larger clusters of even 1,200 memory blocks (*povray*, *roms*). Because some workloads need many separate clusters to achieve substantial compression (e.g., *bwaves* and *cactuBSSN*), hard-coding cluster parameters in hardware is impractical. Finally, because cluster counts and sizes vary widely across the benchmark suite, clustering must be done dynamically at runtime, with performance considerations dictating a hardware-based or hardware-accelerated solution.

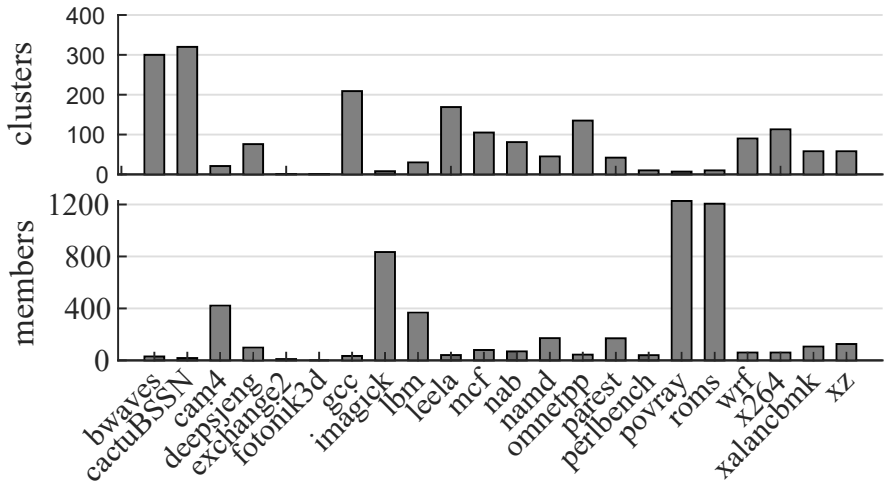



Figure 4.3: Cluster parameters after applying DBScan to LLC snaphots from different SPEC workloads. Workloads were run for 40B instructions after skipping the 100B instructions. Cluster distance was set to achieve an average 40% space savings in each snapshot.

## 4.4 Dynamic Clustering

Directly applying clustering techniques to cache compression is complicated by two challenges. Firstly, cache contents can change as often as every few cycles as lines are inserted and evicted, so there is never a stable image "snapshot" to be analyzed. Secondly, the need to incorporate clustering in a cache controller requires that it is both relatively quick (on the order of a few cycles) and inexpensive to implement in hardware. These requirements exclude common clustering algorithms like DBScan [80].

To overcome these challenges, we observe that an *approximate* clustering technique — one where a point is placed in the "correct" cluster with *high probability*, but can also end up in an entirely "wrong" cluster with *low probability* — is sufficient for cache compression. This is because the few lines that end up in the wrong cluster can simply be stored uncompressed; provided this happens rarely, compression ratio will not be significantly affected.

Thesaurus therefore uses a dynamic approximate clustering mechanism based on locality-sensitive hashing [127]. In this section, we will briefly discuss two key underlying concepts: (a) how locality-sensitive hashing can be

used for approximate clustering, and (b) how locality-sensitive hashing can be efficiently implemented in hardware.

### 4.4.1 Locality-Sensitive Hashing (LSH)

LSH was initially developed as a data structure for the approximate-nearest-neighbour problem [127]. It has been especially popular in big-data environments, and used for streaming nearest-neighbour queries [163, 190], encrypted data search in cloud storage [283], detecting near-duplicate web pages [169], finding DNA patterns [44], unsupervised learning [89], computer vision [65], deep learning [79, 233], etc.

The idea is to create a family of hash functions that map points in some metric space to discrete buckets, so that the probability of hash collision is high for nearby points but low for points that are far away from each other. Specifically, given two points $x$ and $y$ in an $d$-dimensional real space $\mathbb{R}^d$ with a distance metric $\|x, y\|$, a family of hash functions $\mathcal{H}$ is called *locality-sensitive* if it satisfies two conditions:

1. if $\|x, y\| \leqslant r_1$, then $\Pr[h(x) = h(y)] \geqslant p_1$, and
2. if $\|x, y\| > r_2$, then $\Pr[h(x) = h(y)] \leqslant p_2$,

when $h$ is chosen uniformly at random from $\mathcal{H}$, $r_1 < r_2$ are distances, and $p_1 > p_2$ are probabilities [127]. In the context of cache compression, we want a distance metric that (a) correlates with the number of bytes needed to encode the difference between $x$ and $y$, and (b) is easy to evaluate, the $\ell_1$ metric being a natural candidate. Typically, we want $r_2 = (1+\varepsilon)r_1$ for some small $\varepsilon > 0$.

To see how such a hash family could be created, consider the space $\{0, 1\}^d$ of $d$-bit strings under the Hamming distance metric, and, without loss of generality, choose two bit strings $x$ and $y$ in this space. Let $\mathcal{H} = \{h_1, \ldots, h_d\}$, where $h_i$ simply selects the $i$th bit of its input. Intuitively, if $\|x, y\|$ is small (i.e., few bits differ), the probability that $h_i(x) \neq h_i(y)$ (i.e., selecting a different bit) with a randomly selected $h_i$ will be small; in contrast, if $\|x, y\|$ is large (i.e., many bits differ), the probability that $h_i(x) \neq h_i(y)$ will be high. Observe that the difference between the two probabilities will be amplified if we again select an $h_j$ at random and require $x$ and $y$ to match under both $h_i$ and $h_j$ to conclude that $h(x) = h(y)$.

The locality-sensitive hashing algorithm leverages this insight by mapping each point to an *LSH fingerprint* by concatenating the outputs of $k$ randomly chosen functions in family $\mathcal{H}$. Typically, bit sampling is replaced

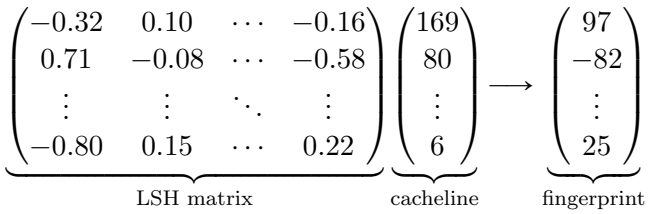



Figure 4.4: Computing the fingerprint of a cacheline using dimensionality reduction.

with multiplication by a matrix randomly sampled from a suitably constructed normal distribution, as illustrated in Figure 4.4; such random projections preserve the distance between any two points to within a small error [83, 137]. By carefully selecting the LSH matrix, an arbitrarily high probability of finding a near neighbour within a chosen radius can be achieved (see [90] for details and a formal analysis).

### 4.4.2 Using LSH for Clustering and Compression

The fingerprint obtained by applying the chosen subset of $\mathcal{H}$ and concatenating the results naturally leads to a clustering scheme where all points with the same fingerprint are assigned to the same cluster. Crucially for cache compression, computing this fingerprint requires no preprocessing or queries of previously cached data.

At the same time, there are two challenges. One is that a cacheline may rarely be assigned to the "wrong" cluster, and be incompressible with respect to that cluster's clusteroid. Because this occurs very rarely, however, the effect on the overall compression ratio is negligible.

The other challenge is that cluster diameters vary across workloads (see section 4.3), but combining LSH functions in a single fingerprint requires fixing the near-distance radius $r_1$ (see subsection 4.4.1). Again, correctness is not compromised because the "misclassified" lines can be stored uncompressed; however, the near-distance radius must be carefully chosen to keep those events rare and provide good compression.

To effect compression, we must also select a base (clustroid) for each cluster: the base will be stored uncompressed, and lines in the same cluster will be encoded as differences with respect to this base (see subsection 4.5.1). Because a clustering scheme based on LSH treats all points in a cluster equally and does not identify a true centroid, we simply choose the first cacheline to be inserted with a given LSH as the cluster base.

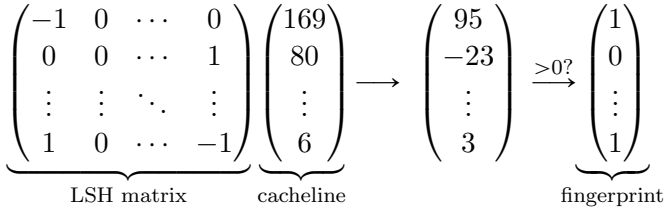



Figure 4.5: A hardware-friendly variant of dimensionality reduction for computing the fingerprint of a cacheline developed for Thesaurus

### 4.4.3 Hardware-Efficient LSH

A key disadvantage of the dimensionality reduction method for computing LSH fingerprints (see Figure 4.4) is that applying the LSH matrix to the cacheline requires many expensive multiplication operations (e.g., 64 if we treat the cacheline as a byte vector); directly implementing this would incur unacceptable overheads in silicon area, latency, or both.

The hardware-efficient LSH implementation in Thesaurus combines two separate refinements of random projection. The first is that multiplication can be avoided by replacing the elements of the LSH matrix with $+1$, 0, or $-1$, chosen at random with probabilities $1/6$, $2/3$, and $1/6$ respectively, with negligible effects on accuracy [17]. Indeed, the sparsity can be further improved by reducing the probabilities of non-zero values to $d/\log(d)$ (where $d$ is the number of dimensions in the original space), again at negligible accuracy loss [160]; this allows for very efficient hardware implementations [82]. (Refer to [17, 160] for a formal analysis of these optimizations.)

To reduce the resulting LSH fingerprints from many bytes to a small number of bits, we combine this with another refinement: the idea that each component of the LSH fingerprint vector can be replaced with 1 if it is positive or 0 if it is negative while retaining the chosen LSH probability bounds [53]. Besides resulting in small fingerprints, this allows us to select the fingerprint size at bit granularity by simply varying the number of hash functions used (i.e., number of LSH matrix rows).

Figure 4.5 illustrates the LSH fingerprint computation in Thesaurus. The cacheline is first multiplied by a sparse matrix with entries from $\{-1, 0, 1\}$; then, only the sign of each scalar is retained, resulting in a fingerprint bit vector. Figure 4.6 illustrates the hardware implementation using only adders and comparators.

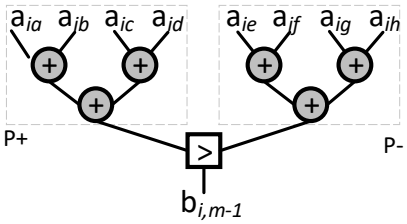



Figure 4.6: Hardware implementation using an adder tree and a comparator.

## 4.5 Cache Architecture and Operation

Briefly, Thesaurus operates by applying the LSH hash as each memory block is inserted. When another "base" memory block with the same LSH exists, the incoming memory block is stored as a byte-level difference with respect to the base (if the difference is small enough to result in compression); if no other memory blocks share the LSH, the incoming memory block becomes the new "base" for the cluster.

Below, we first outline the Thesaurus compression format and storage structures, then walk through an example, and finally detail how Thesaurus operates.

### 4.5.1 Compression Format

Thesaurus uses two primary data encodings: a compressed BASE+DIFF format, and an uncompressed RAW format used when compression is ineffective. We also use secondary data encodings to optimize for three common-case patterns: ALL-ZERO lines, BASE-ONLY for lines that do not need a diff, and 0+DIFF for lines that do not need a base.

In the BASE+DIFF encoding, illustrated in Figure 4.7, memory blocks within a cluster are represented as a compacted byte difference with respect to a base memory block common to the entire cluster. The encoding consists of a 64-bit mask that identifies which bytes differ from the base ❷, followed by a sequence of the bytes that differ ❶. During decompression, the "base" and the compressed "diff" encoding are combined by replacing the relevant bytes ❸.

Lines encoded as ALL-ZERO are identified as such in the tag entry (see section 4.5.2), and require no additional storage. Similarly, BASE-ONLY lines are equal to the cluster base, and so do not need a diff entry in the data array. Finally, 0+DIFF lines are encoded as a byte-difference from an all-zero

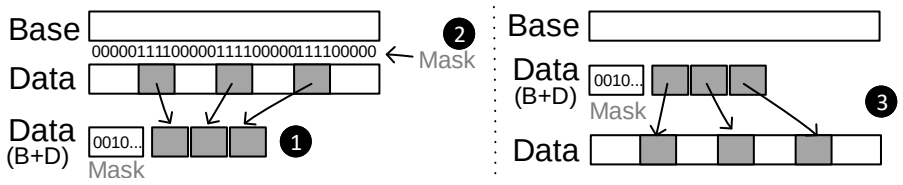



Figure 4.7: The BASE+DIFF compression encoding in Thesaurus. Left: compression; right: decompression.

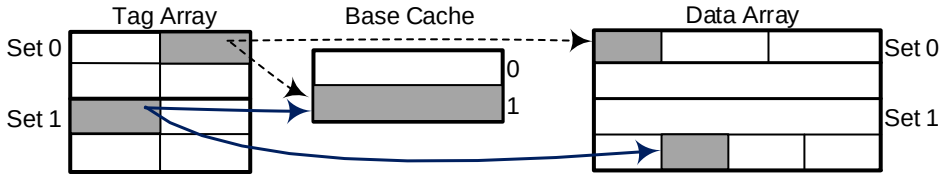



Figure 4.8: A two-set, two-way Thesaurus cache with two memory blocks cached in the BASE+DIFF format. The entries share the same base but have different diffs.

cacheline.

### 4.5.2 Cache Structures

Figure 4.8 shows the storage structures used to implement Thesaurus and the connections among them. As is typical in prior compressed cache proposals [202, 242, etc.], the tag array and the data array are decoupled to enable the storage of a larger number of tags as compared to data entries.

Thesaurus stores the clusteroids for each possible LSH fingerprint in main memory, and caches the most recently used clusteroids within a small LLC-side structure similar to a TLB, which we refer to as the *base cache*.

#### Tag Array

The tag array is indexed by the physical address of the LLC request, and entries are formatted as shown in Figure 4.9. The *tag*, *coh*, and *rpl* fields respectively correspond to the tag, coherence state, and replacement policy state of a conventional cache. The new *lsh* field identifies the LSH fingerprint for the cached data, which points to the clusteroid for this LSH in the base table (see section 4.5.2). The *setptr* field points to a set in the data array, whereas *segix* identifies the segment within the set (see section 4.5.2

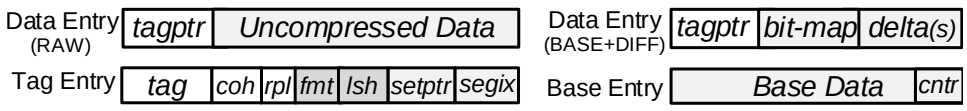



Figure 4.9: Top: Data array entries for uncompressed data (left) and the BASE+DIFF/0+DIFF encodings (right). Bottom: Tag entry format (left) and the base table entry that contains base (right).

for details). Finally, the *fmt* field identifies the cacheline as an ALL-ZERO cacheline, a BASE+DIFF encoding, or an uncompressed RAW cacheline.

### Data Array

As in a conventional cache, the data array is organized in individually indexed sets. To facilitate the storage of variable-length compressed diffs, however, each set is organized as a sequence of 8-byte segments: a single data array entry (i.e., cacheline in BASE+DIFF or RAW format) may take up anywhere from two to eight segments. To avoid intra-set fragmentation, segments in a set are compacted on eviction as in prior work [200, etc.].

As the data array is decoupled from the tag array, any set in the data array can store the incoming memory blocks. Therefore, each data array entry contains a *tagptr* that identifies the corresponding tag array entry. This entry is used to evict the tag if the data array entry is removed to free space for an incoming memory block.

Each set also contains a map called the *startmap*. The startmap helps identify which segments begin new entries; this enables intra-set compaction without the need to traverse the tag array and modify the (possibly many) tag entries to reflect new data locations. The startmap has as many entries as there are segments, where each entry is one of VALID-RAW, VALID-DIFF, or INVALID (128 bits total).

The startmap works in conjunction with the *segix* field in the tag array: segix identifies the *ordinal* index in the set (e.g., first, second, $n$th, etc.), while the startmap identifies which entries are valid. The location of the $n$th entry is obtained by adding the sizes of the first $n-1$ VALID-RAW or VALID-DIFF startmap entries. Because evicted entries can set their startmap tags to INVALID without affecting the segix for the following entries, sets can be recompacted without updating the tag array.

Entries can come in two flavours: RAW and BASE+DIFF. Lines stored in the RAW format (Figure 4.9, top-left) contain a 15-bit tag pointer followed by 64 bytes of data across eight contiguous segments. Lines in the BASE+DIFF

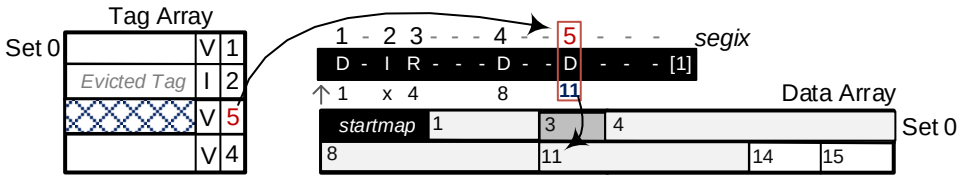



Figure 4.10: The segment index (here, 5) and the startmap combine to locate the compressed data block within a set.The second entry (shaded grey) is invalid, so it is skipped in the startmap. D=VALID-DIFF, R=VALID-RAW, I=INVALID.

format (Figure 4.9, top-right) also begin with a 15-bit tag pointer; this is followed by a 64-bit map that identifies which bytes differ from the base, and then by a sequence of the differing bytes.

#### Base Table and Base Cache

To store the clusteroid (base memory block) for each LSH fingerprint, Thesaurus uses a global in-memory array allocated by the OS, which we refer to as the *base table*. Each base table entry contains a counter of how many current cache entries are using this base. When the counter decreases to 0, the base is replaced with the next incoming cacheline for that LSH, which allows Thesaurus to adapt to changing working sets.

For performance, the base table is cached in a TLB-like table near the LLC, which we refer to as the *base cache*; for us, this is an pseudo-LRU-managed, 64-set, 8-way set-associative structure. The entries in this cache contain the base entry itself, an LSH tag, and replacement policy state.

### 4.5.3 Cache Operation

#### Read Requests

Figure 4.11 shows how Thesaurus services a read request. Shaded areas are on the critical path, while unshaded areas occur after the read has been serviced.

When the request is received, the address is looked up in the tag array as in a conventional cache. If the tag hits, the *setptr* is used to index the data array ❶ (for 0+DIFF and BASE+DIFF formats) and, in parallel, the *lsh* indexes the base cache ❷ (for BASE-ONLY and BASE+DIFF formats). If the LSH is not in the base cache, an access is made to the memory to retrieve the base entry for that LSH (not shown).

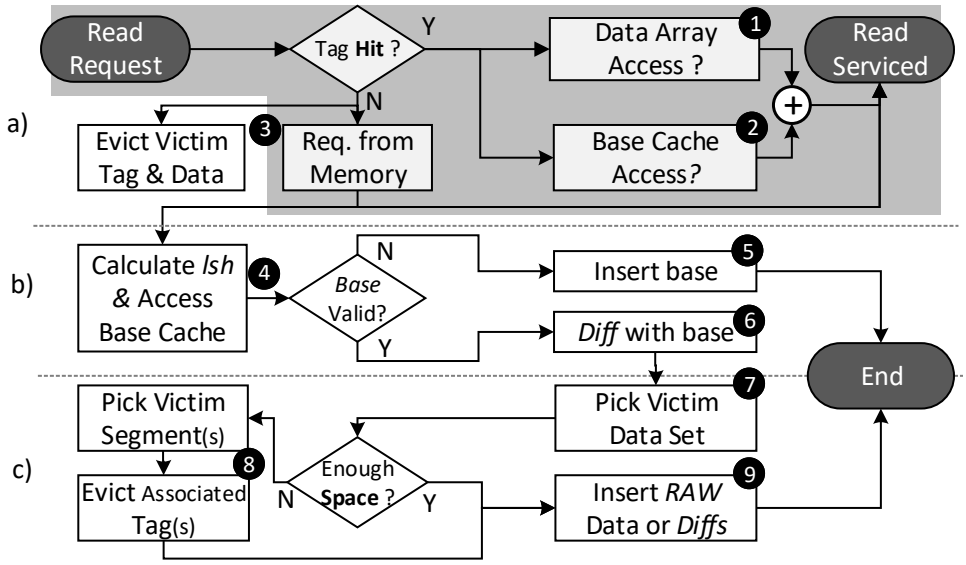



Figure 4.11: Processing a read request in Thesaurus: (a) critical-path lookup sequence; (b) cluster ID computation for new data brought in by a miss; (c) insertion of new data and possible data array evictions. Shaded steps are on the critical path, while unshaded steps are performed in parallel with servicing the read request.

If the tag misses, a request is made to the backing memory ❸; meanwhile, the victim tag and its corresponding data array entry are evicted and the new tag is inserted. The data is returned to the requesting cache as soon as it arrives; inserting the new line in the cache happens off the critical path as other read requests are handled.

If the newly arrived line consists of zeros, an ALL-ZERO tag is inserted and processing ends. Otherwise, the LSH for the newly arrived line is computed ❹, and used to index the base cache; if the LSH is not in the base cache, the data is inserted uncompressed (in RAW format) while the base is retrieved from the base table in memory (not shown). If there is currently no base for the LSH, the new line is installed as the base and processing ends ❺ by inserting a BASE-ONLY tag entry. Otherwise, the byte-difference with respect to the base is calculated ❻; if there are no differences, the a BASE-ONLY tag is inserted and no entries in the data array are made.

For non-base entries, the diff is packed together with a bitmask in the BASE+DIFF or 0+DIFF format; if compression is not possible, the entry will use the RAW format. In either case, the appropriate tag is inserted, and a block must be added to the data array. To make space, a data array victim set is selected ❼ as described in section 4.5.3; if there is not enough space there, enough victim segments are selected to make space for then new block, and their tags evicted ❽. Finally, the block is inserted, possibly recompacting the set ❾.

### Write and Atomic Requests

Accesses that modify the data can change the mode of compression or the size of the compressed block. If the diff is smaller, the data array entry is either removed (for ALL-ZERO or BASE-ONLY) or the block's bitmap is updated and the set is compacted. If the diff is larger, other entries are evicted from the set to make space ❽, and the set is updated and compacted.

For write operations in memory models without write acknowledgements (most extant ISAs), the entire write is performed off the critical path. For atomic read-modify-write operations (e.g., compare-and-swap), the read part is serviced as soon as possible, and the write part is completed off the critical path.

### Replacement Policies

The tag array follows the corresponding conventional replacement policy (in this work, we use pseudo-LRU); the base cache follows pseudo-LRU. Unlike

in a conventional cache, however, the data array entry requires a separate replacement policy: in this case, a policy that favours evicting fewer data entries over recency makes sense, as not-recently-used data array entries will have been evicted anyway by the tag array replacement policy.

To choose a victim set, we use a best-of-$n$ replacement policy [88, 242]. First, we randomly select four sets. If one of the sets has enough free segments to store the incoming (possibly compressed) block, it is chosen and no evictions are made; otherwise, we select the set with the fewest segments that would have to be evicted to make enough space.

Observe that the randomness ensures that frequently used blocks do not evict each other in a pathological pattern: if a block is evicted and soon thereafter reinserted, it will likely end up in a different set than the block that evicted it.

### 4.5.4 Walk-through Examples

Figure 4.12(b) shows an example lookup the Thesaurus LLC. First, as in a conventional cache, the address is used to index the tag array ❶; in this example, the cacheline uses the BASE+DIFF encoding. The LSH stored in the tag entry is used to index into the base cache and retrieve the compression base ❷. Concurrently, the set index from the tag entry is used to retrieve the set. The segment index from the tag entry and the startmap from the data entry are combined to identify the beginning of the encoded cacheline in the set ❸. Finally, the bitmask and bytes stored in the diff entry are used to replace the corresponding bytes from the base entry, and the resulting line is returned ❹.

Figure 4.12(c) illustrates how the startmap is updated during an eviction, showing a set before and after evicting entry *d1*. Before the eviction, *d1* is the second VALID index in the startmap ❶, *d2* is the third, and so on. After the eviction, the tag array entry for $d1$ has been invalidated ❷, the other entries (*d0*, *d2*) have been compacted to form a contiguous set, and the startmap entry that previously identified $B$ has become INVALID ❸. This means that *d2* is still the third overall entry, and the tag array entries for *d2* do not need to be updated to reflect the compaction.

Finally, Figure 4.12(d) shows an insertion of a new entry *d3*. First, the access misses in the tag array and a request is made to the backing memory ❶. When the cacheline data arrives, it is immediately returned to the requesting cache (e.g., L2) ❷. In parallel, the cacheline's LSH fingerprint is computed ❸ and used to index the base cache. In our example this access hits and returns the data for the base ❹. The incoming line is then XOR'd

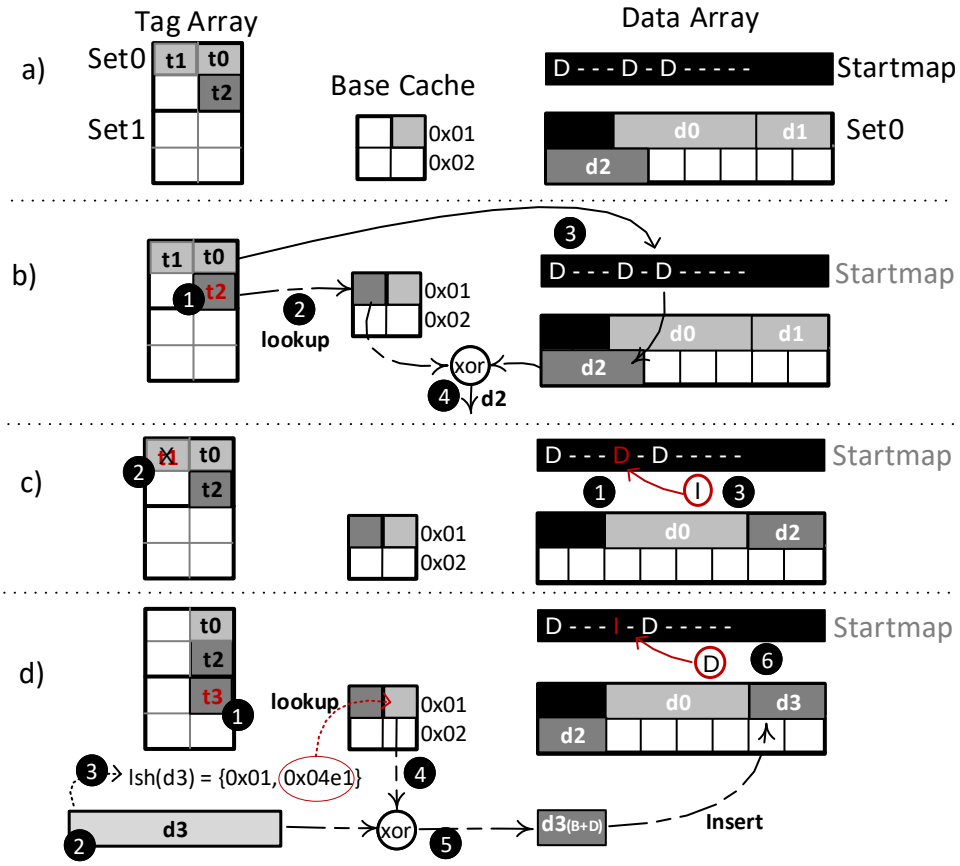



Figure 4.12: Thesaurus structures during cache operations: (a) initial state; (b) read request processing; (c) eviction; (d) new entry insertion.

| | |
|---|---|
| CPU | x86-64, 2.6GHz, 4-wide OoO, 80-entry ROB |
| L1I | 32KB, 4-way, 3-cycle access lat., 64B lines, LRU |
| L1D | 32KB, 8-way, 4-cycle access lat., 64B lines, LRU |
| L2 | Private, 256KB, 8-way, 11-cycle lat., 64B lines, LRU |
| LLC | Shared 1MB, 8-way, 39-cycles lat., 64B lines, 8 banks |
| Memory | DDR3-1066, 1GB |

Table 4.1: Configuration of the simulated system.

with the base, and the non-zero bytes of the resulting difference are encoded as a bitmask and a list of differing bytes ❺; in the example, this encoding takes up 16 bytes, or two segments. Next, this entry is inserted in the cacheline from the previous example containing *d0*, *d2*, and *d3*: the INVALID startmap entry is replaced by VALID-DIFF ❻, and the entry is inserted in the corresponding sequence in the set ❼.

## 4.6 Methodology

To evaluate effects on cache behaviour and performance, we implemented Thesaurus and the comparison baselines in the microarchitecture-level simulator ZSim [213]. We simulated an out-of-order x86 core similar to an i5-750, modelling on- and off-critical-path events as well as limited interconnect bandwidths; the simulated system is shown in Table 4.1. Compression was applied at the LLC level only.

To estimate silicon area and power impacts, we implemented all logic that is required in Thesaurus but not in a conventional cache in Verilog RTL, and synthesized these with Synopsys Design Compiler tool using FreePDK45 [236] standard cell library. We used CACTI 6.5 [181] to estimate the area, access time, and power of storage structures.

As the baseline, we modelled a conventional (uncompressed) LLC with 1MB capacity per core; we also modelled a hypothetical LLC with 2× the capacity (2MB), which has an effective capacity similar to a 1MB Thesaurus cache. To study relative improvements over prior state-of-the art intra- and inter-cacheline compression, we also implemented B$\Delta$I [200] and Dedup [242]. All compressed configurations (Thesaurus, Dedup, B$\Delta$I) were sized to occupy the same silicon area as the baseline LLC: Table 4.6 compares the storage allocations.

To find a suitable LSH size, we swept sizes of 8–24 bits. We found that 12-bit LSHs result in good compression for most workloads while keeping

| | | Conv. | BΔI | Tian:2014:LLCDedup | Thesaurus |
|---|---|---|---|---|---|
| **Tag** | #Entries | 16384 | 32768 | 32768 | 32768 |
| | Entry Size | 37b | 47b | 81b | 72b |
| | Total Size | 74KB | 188KB | 324KB | 288KB |
| **Data** | #Entries | 16384 | 14336 | 11700 | 11700 |
| | Entry Size | 512b | 512+0b | 512+16b | 512+32b |
| | Total Size | 1024KB | 896KB | 754KB | 777KB |
| **Dict.** | # Entries | - | - | 8192 | 512 |
| | Entry Size | - | - | 24b | 24+512b |
| | Total Size | - | - | 24KB | 33KB |
| | **Total Size** | **1.07MB** | **1.06MB** | **1.07MB** | **1.07MB** |

the base table size low.

We evaluated all designs on the SPEC CPU 2017 suite [43]. For each benchmark, we skipped the first 100B instructions, and executed the last 20% of each 1B instructions. For miss rate and speedup measurements, we split the benchmarks into cache-sensitive (S) and cache-insensitive (NS). In our evaluation, a benchmark was considered cache-sensitive if doubling the cache size to 2MB improves the MPKI by more than 10%. (In a practical implementation, the LLC could dynamically detect cache-insensitive workloads by measuring average memory access times and disable LLC compression.)

## 4.7 Evaluation Results

### 4.7.1 Compression Analysis

Figure 4.14 shows the improvements over the uncompressed baseline and state-of-the-art compressed caches. We also compare against the ideal clustering method (which searches the entire cache for the nearest match and diffs against it in one cycle) and a conventional cache with 2× the capacity.

Figure 4.14(a) shows the effective cache footprint reduction — i.e., the data array space taken up by the cached addresses normalized to the equivalent space that would have been required to hold the same addresses in a conventional cache. Overall, Thesaurus compresses the working sets by 2.25×, compared to 1.28× for exact deduplication and 1.48× for BΔI compression (all geomeans); this demonstrates that Thesaurus compresses more effectively than state-of-the-art cache compression techniques.

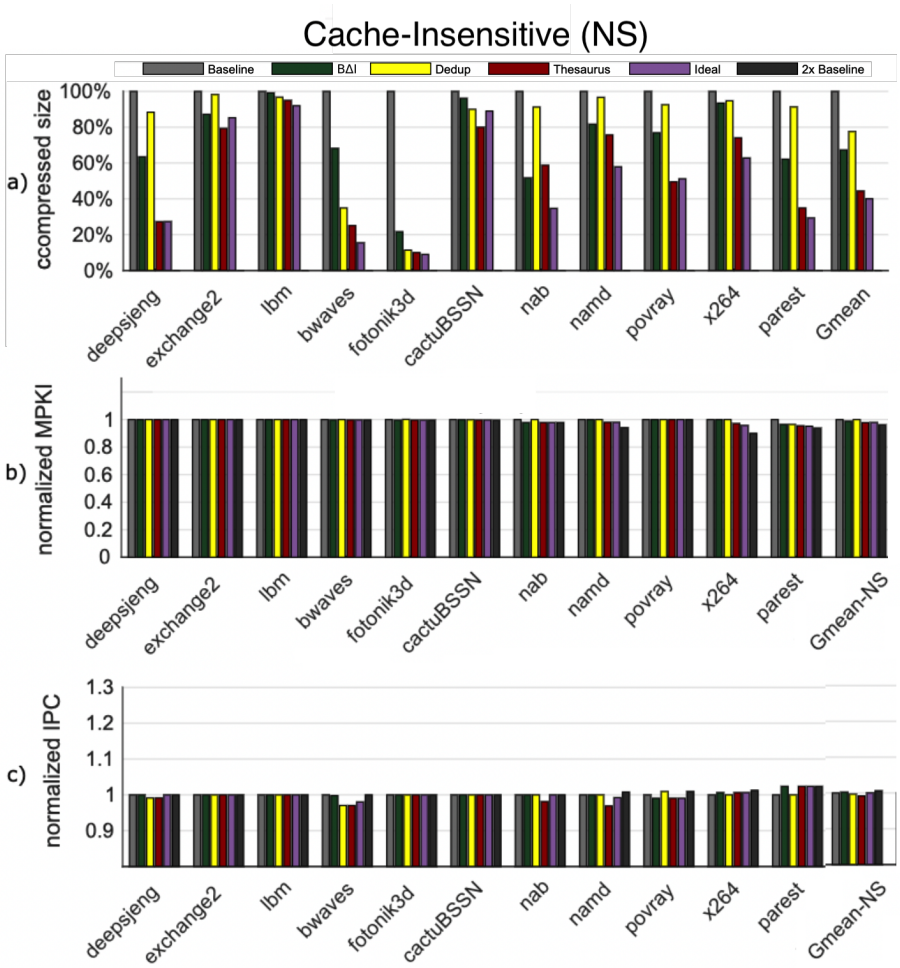



Figure 4.13: Compressed working set size, cache miss rates, and performance of Thesaurus compared to baseline (uncompressed) cache, iso-silicon BΔI (intra-block) and Dedup (inter-block), Ideal-Diff, and an uncompressed cache with 2× capacity on cache-insensitive benchmarks. All but ideal and the 2× baseline are sized to the silicon area of the uncompressed cache. a) Average cache occupancy (100% = no savings,0% = perfect compression), b)Misses per 1,000 instructions (MPKI) relative to a conventional (uncompressed) cache (lower is better), c)Performance improvement normalized to a conventional (uncompressed) cache.

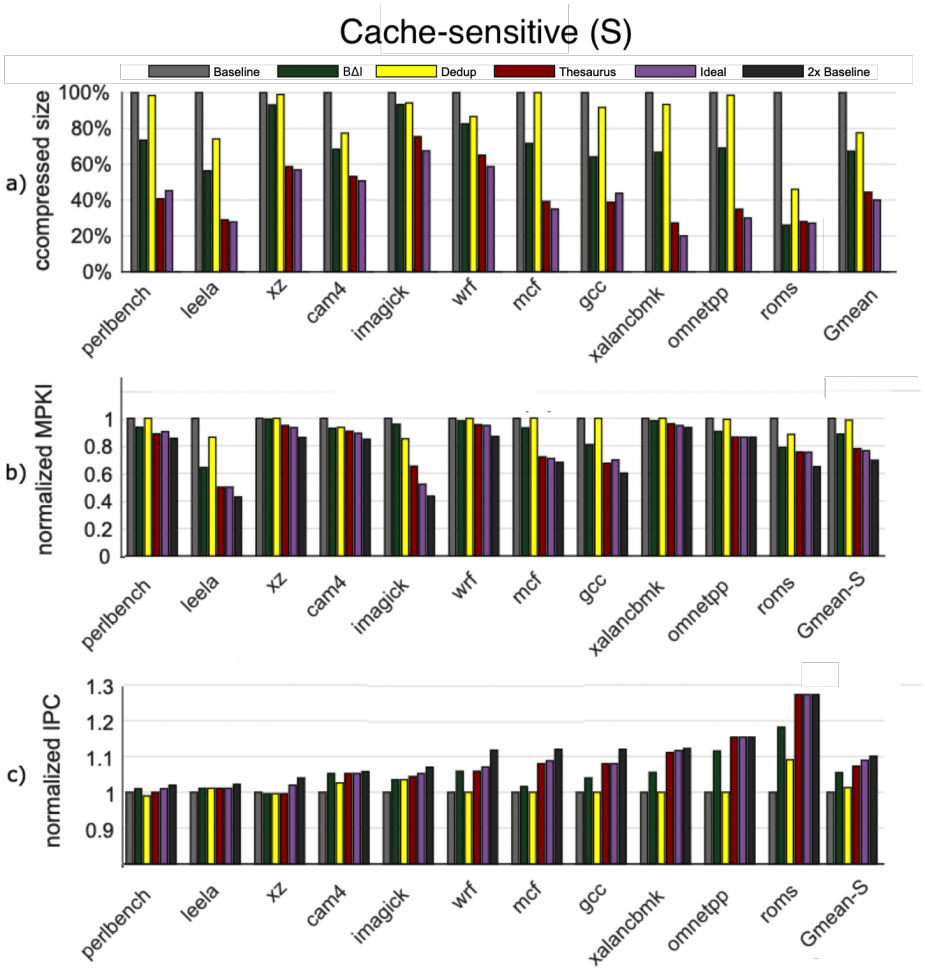



Figure 4.14: Compressed working set size, cache miss rates, and performance improvements of Thesaurus compared to baseline (uncompressed) cache, iso-silicon BΔI (intra-block) and Dedup (inter-block), Ideal-Diff, and an uncompressed cache with 2× capacity on cache-sensitive benchmarks. All but ideal and the 2× baseline are sized to the silicon area of the uncompressed cache. a) Average cache occupancy (100% = no savings,0% = perfect compression), b)Misses per 1,000 instructions (MPKI) relative to a conventional (uncompressed) cache (lower is better), c)Performance improvement normalized to a conventional (uncompressed) cache.

The LSH scheme in Thesaurus also captures nearly all data that can be effectively clustered: the cache footprint obtained using Thesaurus is within 5% of the ideal clustering scheme, which searches the entire cache for the nearest match. In a few cases, compression is, in fact, slightly *better* (e.g., *povray*, *perlbench*, *gcc*): this is because Thesaurus can diff against a clusteroid whose tag has since been evicted from the cache, while the ideal clustering model is restricted to currently cached entries.

### 4.7.2 Miss Rates Analysis

Figure 4.14(b) shows that Thesaurus substantially reduces miss rates: for the cache-sensitive subset of the suite, MPKI drops to 0.78 of the conventional cache compared to 0.98 for exact deduplication and 0.89 for B$\Delta$I (all geomeans). Thesaurus is also within 1.5% of the ideal clustering model, and within 8% of the MPKI that can be attained with a conventional cache with 2× capacity. This is because, thanks to the effective compression, more data can be cached within the same silicon area, which benefits cache-sensitive workloads.

### 4.7.3 Speedup Analysis

Figure 4.14(c) shows that the reduced MPKI rates result in execution time speedups over the baseline as well as Dedup and B$\Delta$I. Thesaurus is up to 27.2% faster than the conventional baseline (7.9% geomean), and up to 9.1% faster than B$\Delta$I (5.4% geomean). Indeed, performance is within 1.1% of the ideal clustering model, and within 2.2% of a conventional cache with 2× capacity.

### 4.7.4 Cost Analysis

**Latency.** We modelled access times to each LLC structure using CACTI; because compressed caches have smaller data array sizes, its overall access time is slightly reduced (∼2% for Thesaurus). To measure compression and decompression latencies, we implemented the logic for compression, decompression, as well as locating and reading the compressed cachelines in 45nm ASIC; the results are shown in Table 4.3. At the relevant CPU frequency (2.66GHz), compression and decompression take one cycle each, while locating the compressed data block in the set (described in section 4.5.2) takes four more cycles. This brings the total decompression latency to 5 cycles, which we used for the performance simulations.

| | 45nm | | 32nm | |
|---|---|---|---|---|
| | dynamic energy | leakage power | dynamic energy | leakage power |
| Conv. | 0.50 nJ | 205.47 mW | 0.28 nJ | 109.96 mW |
| B$\Delta$I | 0.55 nJ | 196.47 mW | 0.31 nJ | 105.22 mW |
| Dedup | 0.56 nJ | 226.33 mW | 0.32 nJ | 121.06 mW |
| **Thesaurus** | **0.56 nJ** | **236.01 mW** | **0.31 nJ** | **125.85 mW** |
| Conv. 2× | 0.78 nJ | 349.21 mW | 0.44 nJ | 186.50 mW |

Table 4.2: Dynamic read energy and leakage power per bank of compressed and conventional caches scaled to the same silicon area (1MB uncompressed = 5.56mm$^2$ in 45nm or 2.82mm$^2$ in 32nm).

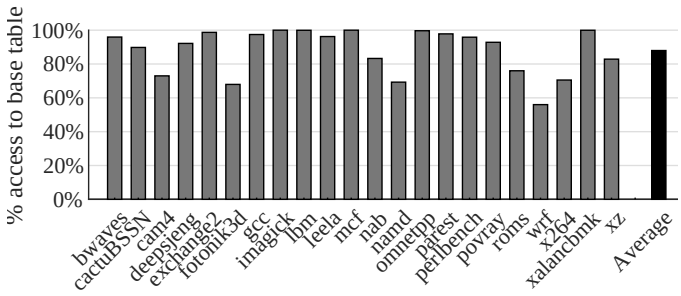



Figure 4.15: Fraction of cache insertions that are potentially compressible with respect to their clusteroid (avg. 87%).

**Power.** We used CACTI 6.5 [181] to estimate the read energy and leakage power of all cache structures; the results are shown in Table 4.2. While Thesaurus uses ∼12% more energy for each read and has a ∼14% leakage power overhead, these overheads are significantly lower than ∼57% increase in dynamic energy and ∼70% increase in leakage power for conventional cache with the same effective capacity.

Most importantly, the overhead of Thesaurus (∼0.06nJ per access at the 45nm node) is trivial compared to the energy of accessing external DRAM (32.61nJ using the same CACTI model). This means that Thesaurus can actually *save* energy when the entire memory hierarchy is considered. In order to measure this, we calculated the total added power of compressed cache (30.54mW + 6.4mW + 0.06nJ × access rate) and total power saved by accessing off-chip DRAM less frequently (32.61 nJ × access rate difference between Thesaurus and uncompressed).

| | latency | dynamic power | leakage power | area |
|---|---|---|---|---|
| comp. | 1 cycle | 0.116 mW | 2.44 mW | 0.016 $mm^2$ |
| decomp. | 1 cycle | 0.084 mW | 1.74 mW | 0.013 $mm^2$ |
| segix | 4 cycles | 0.035 mW | 0.49 mW | 0.007 $mm^2$ |
| multi-bank | - | 0.101 mW | 1.42 mW | 0.025 $mm^2$ |

Table 4.3: Synthesis results for the added logic area of Thesaurus: *segix* refers to locating the compressed block within a set (decoding the indirect segix format), while *multi-bank* refers to the muxing needed to access lines across multiple banks; 64-byte cachelines were used. Latency is in units of CPU cycles at the 2.66GHz frequency of the equivalent 45nm i5-750 core. All results obtained Synopsys DC and the 45nm FreePDK.

While the power consumption of the Thesaurus LLC increases (from 36.87mW to 51.28mW) because of the added logic and more data array reads (due to the higher LLC hit rate), accounting for DRAM accesses results in power consumption savings of up to 101mW in the cache sensitive benchmarks. Cache-insensitive benchmarks do not see fewer off-chip DRAM accesses despite effective compression, and therefore power overheads are not outweighed by power savings; however, a practical implementation would detect cache-insensitive workloads and simply disable compression for cachelines they access.

**Area.** We implemented and synthesized the compression and decompression units in Thesaurus. The logic required for Thesaurus incurs an area overhead of 0.06mm$^2$ in 45nm: this includes the compression (0.016mm$^2$) and decompression (0.013mm$^2$), the logic to locate the segments in the set using the indirect segix encoding (0.007mm$^2$), and the additional muxing needed to read a set across multiple banks (0.025mm$^2$). This is equivalent to 1% of the silicon area required for even a 1MB cache, and a tiny fraction of a 4-core i5-750 in the same 45nm node (296mm$^2$).

To compare with prior compressed caches, we also implemented and synthesized the B$\Delta$I scheme [200], which at 0.037mm$^2$ (20k NAND gates) is slightly smaller than Thesaurus (0.06mm$^2$ or 32k NAND gates); this is not surprising as B$\Delta$I offers much less compression. We also used NAND-gate estimates for prior work from [56, 243] to compare against other prior

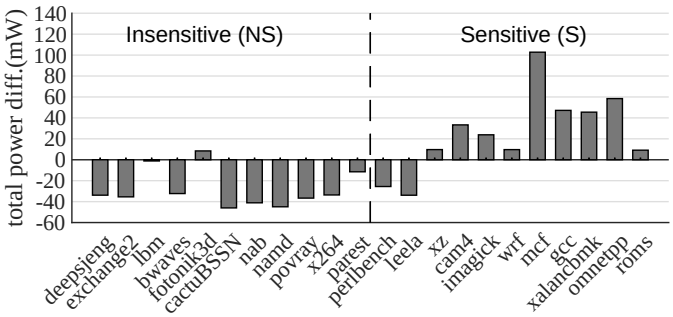



Figure 4.16: Difference in total power consumption using Thesaurus compared to the baseline. Positive values indicates less power consumption whereas negative values shows additional power consumption.

compression schemes, all of which incur more area overhead than Thesaurus: C-PACK [56] needs 0.075mm$^2$ in 45nm (40k NAND gates) and FPC [21] needs 0.544mm$^2$ (290k NAND gates) just for decompression [56], while BPC [143] needs 0.127mm$^2$ (68k NAND gates).

As described in

, all compressed caches are sized to fit in the same silicon size of a 1MB conventional cache. Therefore, overall, 2DCC incurs no area overheads.

### 4.7.5 Clustering Analysis

To investigate the effectiveness of cacheline clustering based on locality-sensitive hashing, we first examined how many cache insertions are, in fact, compressed. Figure 4.15 shows that, on average, 87% (a significant majority) of cache insertions can potentially result in compression — i.e., their differences versus the relevant clusteroid are small enough that encoding them would take $< 64$ bytes even with the overhead of the difference bitmask. This observation validates both our choice of LSH for clustering and the effectiveness of our clusteroid selection strategy to increase compressibility.

Figure 4.17 shows that most working sets have many small clusters rather than few large clusters. Nevertheless, because Thesaurus tracks clustroids for many LSH fingerprints, effective compression can still be obtained.

Next, we examined the compression encodings used by Thesaurus. Figure 4.18 shows that different workloads tend to benefit from different encodings. For most of the benchmarks, the byte-difference-based encodings are the most effective: the BASE+DIFF encoding covers an average of 76.2% of LLC insertions, while 0+DIFF covers a further 13.2%. Another 6.1% are

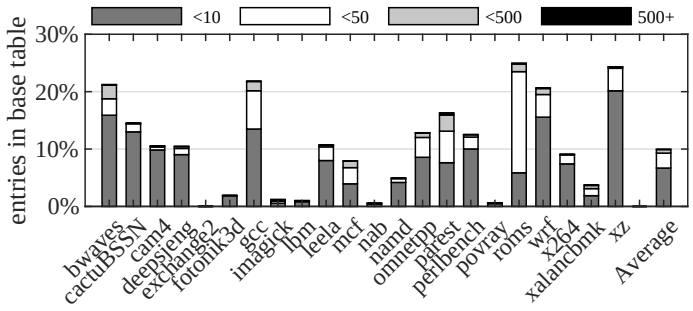



Figure 4.17: Distribution of clusters (= same LSH) with different sizes (average over the runtime of each benchmark).

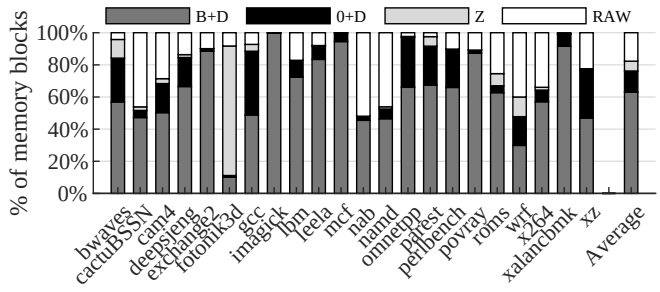



Figure 4.18: Frequency of different compression encodings in compressing benchmarks from SPEC. B+D=BASE+DIFF; 0+D=0+DIFF; RAW=uncompressed; Z=ALL-ZERO.

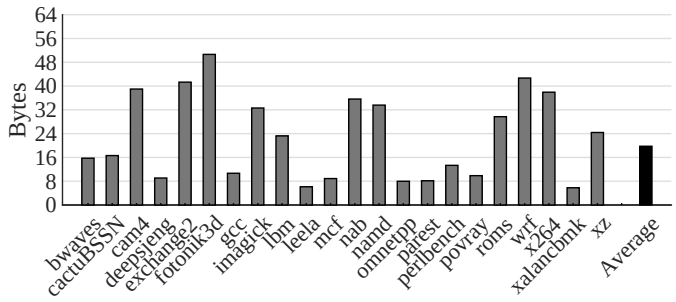



Figure 4.19: The average size of the byte difference from the relevant clusteroid for BASE+DIFF and 0+DIFF, in # bytes.

covered by the ALL-ZERO encoding. Finally, 17.7% of LLC insertions are left uncompressed as the diff from the clusteroid (base) is too large.

Figure 4.19 shows the average size of the byte-difference block for the BASE+DIFF and 0+DIFF encodings. In most cases, the differences tend to be small, with a quarter of the benchmarks averaging about 8 bytes or less, and half of the benchmarks averaging about 16 bytes or less. This confirms that, for many benchmarks, caches have relatively tight data clusters with very small intra-cluster differences even at cacheline granularity.

In fact, diff sizes can change significantly over the run time of even a single workload. Figure 4.20 shows the diff sizes for 1 million cache insertions after the first 100 billion instructions for four workloads. In *mcf*, the data compression ratios are relatively stable; in *xalancbmk*, the tiny diffs of most accesses are punctuated by rare spikes of 32-byte diffs; *bwaves* has two distinct but small diff sizes; finally, *cam4* intersperses blocks that offer little compression with periodic short bursts of compressible data.

Together with the number of insertions that can be compressed (Figure 4.15), the diff sizes explain the compression ratios for each benchmark. For example, more than 90% of inserted blocks in *imagick* can be compressed, but the average diff size is 32.6 bytes, resulting in a compression ratio of 1.3× (see Figure 4.14). In contrast, *xalancbmk* and *mcf* combine a high (> 90%) proportion of compressible insertions with a small average diff size (6 and 9 bytes, respectively), for a total compression factors of 2.6× and 3.7×.

Finally, we established an efficient size for the base cache and examined its effectiveness. To establish size, we swept sizes ranging from 32 to 2048 entries; results are shown in Figure 4.21. Compared to a 94.8% hit rate for a 512-entry cache (33KB), a 2048-entry cache (+100KB storage) increases

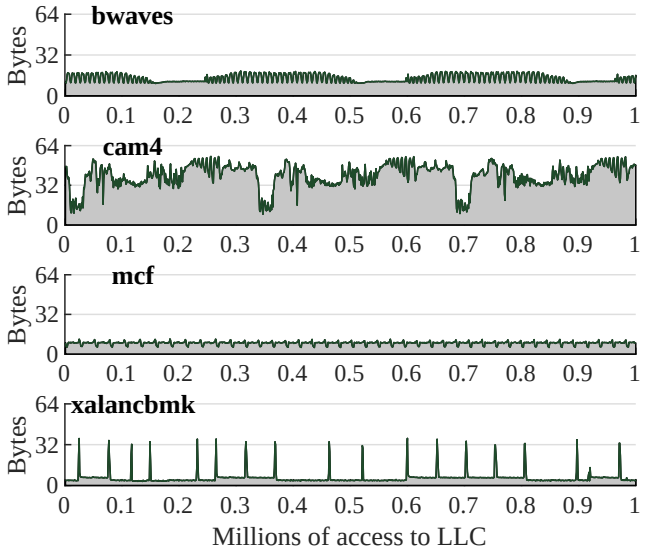



Figure 4.20: How the diff size varies over time: 1 million cache insertions after skipping the first 40B instructions.

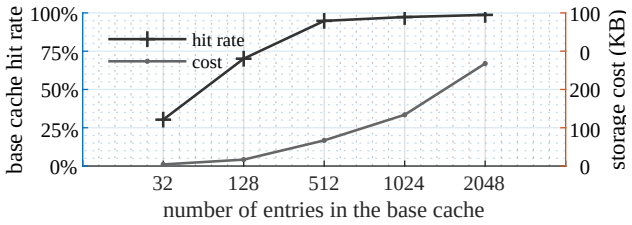



Figure 4.21: Base cache hit rate (left axis) and storage cost (right axis) for different base cache sizes.

the hit rate by only 3.9%; we therefore used a 512-entry base cache for the remainder of our experiments.

On average, this cache has a 5.2% miss rate over all benchmarks; however, all but 8% of misses (i.e., all but 0.5% of accesses) miss when a line is being *inserted* in the cache, and are off the critical path. Because the data is inserted uncompressed while the clusteroid (base) is fetched into the cache, these misses represent a missed compression opportunity but do not affect insertion latency.

Nevertheless, the lost compression opportunities due to base cache misses can have a significant impact. The benchmarks with high off-critical-path base cache miss rates — *bwaves*, *nab*, *namd*, *x264* and, *wrf* with 8–13% — also lose the most compression opportunity compared to the idealized *Ideal-Diff* clustering (cf. Figure 4.14). For those workloads, a larger base cache would improve compression.

### 4.7.6 Threats to Validity

Throughout this chapter and chapter 3, we illustrated how considering the data throughout the cache, as well as leveraging the similarity among the data lines, can improve the compression ratio of the data being stored in the cache and as a result improve the entire system performance (speedup and energy).

For consistency with prior work [20, 21, 200, 242, 260], our evaluations have modeled an Intel Lynnfield core i5-750-like CPU [4] based on the Nehalem microarchitecture [5, 12] with 4-wide OoO and 80 entry ROB, manufactured in 45nm and connected to a DDR3-1066 RAM. Below, we argue that our insights apply to more recent devices, such as Intel Sunny Cove microarchitecture [6] at 10nm (352-entry ROBs) and later.

**Reorder-buffer size**: Any architectural changes and designs comes with a trade-off. While in general increasing the number of ROBs, can be a way to reduce cache miss penalties, on the other hand, larger ROB entries increases the demand for more storage space in the silicon; therefore, to be able to keep our cache footprint within 1MB iso-silicon requirement, we need to trade-off some other memory silicon or some logic silicon (with far less silicon footprint than memory units).

**Main memory technology:** In general, technology advancement not only impacts the area and power consumption, but also impact the latency and access times of CPU and entire memory system. However, using better technology nodes, does not have any effect on the number of unavoidable more expensive off-chip memory accesses (i.e. LLC misses). Cache compres-

sion is trying to reduce the negative impact of missing a block by reducing the number of these misses seen at LLC.

On the one hand, better technology means miss penalties will become less severe due to faster main memories, but on the other hand, better technology also means having even faster CPUs(from 2.6GHz in i5-750 [4] to 3.7GHz in i5-12600 [1]). The effective DRAM access latency in terms of the *number of cycles*, then, tends to actually grow [1].

Overall, while the final speedup and power numbers reported for all caches including the compressed and uncompressed will likely change due to technology advancement, our techniques and insights are independent from the technology being used. As our designs outperforms all other designs in terms of reduced misses, the relative benefits versus prior compressed cache designs should still hold.

**Replacement policy:** While our uncompressed baseline uses LRU replacement policy, we have reimplemented other baselines [200, 242] with their proposed replacement policies (i.e. modified LRU in [200] and DFRR in [242]). In both 2DCC and Thesaurus we have developed a customized replacement policy based on two pillars: a) a timing aspect that accounts for the frequency of accessing the blocks, and b) a compression-aware aspect which takes the compressibility and size of the blocks being replaced, which is orthogonal to the first aspect.

In theory, using a better and state-of-the-art replacement policies (e.g., SHiP[252] and RRiP [134]) can improve the first pillar (impacting the uncompressed baseline's speedup the most). However, these policies are not aware of how compressed caches organize data (e.g., with independent tag and data arrays), and redesigning them to apply to compressed caches is not a trivial task. In addition, the benefits from a better replacement policy mainly affect the first pillar (timing), and are orthogonal to the second pillar (fitting more data in the cache).

Our results demonstrates that the second pillar used in our policy is effective in managing the cache, outperforming all other compressed designs and, therefore incorporating it into another timing-based policy would likely still result in better result that other designs.

**Capturing workload phases**: Understanding the cycle level behavior of a processor running an application is crucial to computer architecture research. Simulating the full execution of industry standard benchmarks in a detailed (cycle accurate) simulation takes on the order of weeks to months to complete. This problem gets even worse as to properly perform an architectural evaluation requires these benchmarks to be evaluated across many separate runs. To address this issue a common practice is to use a

sampling technique to gather simulation data.

There are several works [63, 228] that address this issue in a more sophisticated manner for multi-core systems. SimPoint [228] tries to automatically find a small set of Simulation Points to represent the complete execution of a program by intelligently chosen sections from the full program. An assumption of this work is that most of the important metrics of an application are a function of the code executed, and that IPC is the key figure of merit. This assumption makes it unsuitable for us, as IPC does not convey information about data compressibility. Compressibility of data in an application is generally a slowly changing value, rather than changing in phases like IPC [63].

CompressPoints [63], addresses the limitation of [228] by choosing a region in the application that is representative of both compression ratio and workload performance. Its benefits are more prominent when simulating multiple workloads in a multi-core system with presence of compressed main memory. Instead of capturing representative points individually for each benchmarks, which can miss the fact that these workloads can compete for resources, CompressPoints accounts for interplay between the benchmarks running together.

By skipping the early instructions that fast forwards initialization phase of workloads and then sampling afterwards, we were also able to capture these kinds of workload phase changes. Figure 4.20 illustrates compression phase changes in several benchmarks. Therefore, we believe our methodology is effective in capturing both IPC and compressibility aspects of individual workloads on particular benchmarks that we evaluated.

**Other workloads**: We chose SPEC 2017 benchmarks [43] for our evaluations as they are representative of modern day workloads. They cover wide range of application areas such as finance, database, AI, and so on, and use different data structures and algorithms that are widely used. The benchmark set also consists of variety of cache sensitive and insensitive workloads with different LLC MPKIs; this helps us evaluate our work on various workload scenarios.

We also applied our technique to other database and graph benchmarks such as the GAP Benchmark Suite [34] and observed its effectiveness in reducing cache memory footprint of these benchmarks. Most GAP benchmarks have great locality and very low MPKI, especially those processing the web [35].

As we only run each workload in isolation on a single core setup, cache-insensitive workloads will not benefit from increased cache space, and in come cases may experience minor performance degradations. However, in

a multi-core setup with larger caches, we expect to see considerable benefits: as cache-insensitive workloads can be compressed significantly, running them along other cache-sensitive workloads which convert this extra available cache space to performance, will improve the overall system performance.

## 4.8 Summary

In this chapter, we demonstrated that inter-cacheline compression can be very effective if small differences among the cached memory blocks are allowed. We proposed Thesaurus, an LLC compression based on dynamic cluster detection, and describe an efficient in-hardware implementation based on locality-sensitive hashing. We also develop a replacement policy for the data array in the LLC that balances the development of new clusters with conserving existing clusters.

# Chapter 5

# Dynamic Clustering of Layer Activations in DNNs

In this chapter, we leverage the insights developed for clustering data blocks in CPU caches from chapter 4 to reduce the memory footprint of DNNs. We demonstrate the effectiveness of data clustering in compressing special-purpose applications by designing a dynamic clustering method to tackle activation maps compression in deep neural networks (DNNs).

First, After a brief overview of DNNs, we review the existing compression work on DNN and identify their drawbacks. Then, we show that activation distributions are similar across large groups of channels, and therefore channels can be clustered based on their statistical properties. We propose Channeleon, an activation compression technique for off-the-shelf, optimized, deployment-ready models — all without relying on access to training data, knowledge of the training algorithms and their hyperparameters, or statistics from the batch-norm layers. We continue this chapter by arguing that non-uniform quantization method is necessary to retain the model accuracy as low-bit-widths.

Channeleon maintains accuracy at bitwidth ranges where prior methods fail: it can compress activations to as few as 4 bits, where existing post-deployment methods fail below 8 bits. Channeleon results in 1.33× the compression factors achievable by the 8-bit prior art at the similar accuracy levels.

## 5.1 Deep Neural Networks

Throughout the thesis, we use convolutional neural networks (CNNs) [112, 216, 230, 241, 273, 281] to evaluate our work. CNN is class of deep neural networks (DNNs), most commonly applied to analyze visual imagery. There are two phases involved in employing a DNN model: training and inference.

In training phase, DNN weights are optimized via approaches such as stochastic gradient descent (SGD) [154] using a training dataset in order

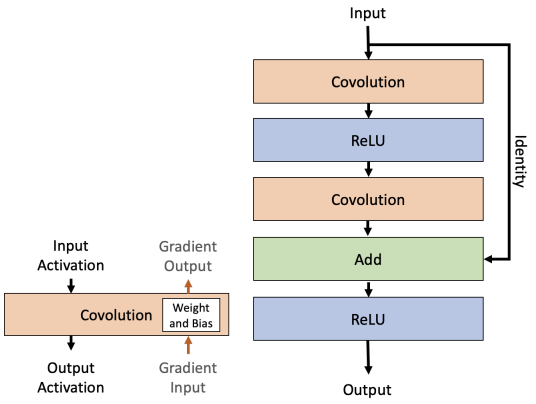



Figure 5.1: (a) A block representing convolutional layer and its activation, weight, bias, and gradient values. (b) A demonstration of network layers for building block of residual learning networks as in Figure 2 and Figure 3 of [112] (black arrows illustrate activation in forward path).

to minimize a certain loss function. In each training step, a forward pass is first performed to calculate the loss, followed by a backward pass to back-propagate the error. Finally, the gradient of each parameter is computed and accumulated. Then, in the inference phase, only forward passes are performed in order to make predictions.

DNNs are designed using a combination of multiple types of layers, most notably convolutional (CONV) layers, fully-connected (FC) layers, activation layers, batch normalization layers, and so on. Depending on the type of a layer, it may consist of various values such as activation, weight, bias, and gradients. Weights and biases are the learnable parameters of a layer, while activations and gradients are the intermediate values that needs to be exchanged among layers. Learnable parameters do not change after the training task is completed and therefore, they can be optimized statically. whereas activation map depend on the input data and therefore, change dynamically. We illustrate activation maps, weights, and other values of a convolution layer in Figure 5.1(a).

CONV layers contains a set of filters to identify meaningful features in the input data. For visual data such as images, usually 2-dimensional filters are employed which slide over the input of a layer to perform the convolution operation. Activation layers apply an element-wise activation function to the input feature maps. The ReLU activation function [185] in particular is known to provide state-of-the-art performance for CNNs, which allows

positive input values to pass through while thresholding all negative input values to zero [27]. For example, we illustrate the layers within the building block of a widely used family of networks for image classification task, ResNet [112], in Figure 5.1(b).

Running DNNs requires lots of computing power and memory and these costs have increased as CNNs get wider and deeper to perform better predictions for a variety of applications [144]. Although a DNN may include many types of layers, matrix multiplications and convolutions dominate over 80% of the operations, and are the main targets of DNN computation acceleration [267]. As mentioned in chapter 2, model compression methods [32, 48, 61, 62, 95, 106, 164, 184, 207, 253] try to reduce the memory footprint as well as computation requirement of DNNs by removing or estimating layer parameters and activations.

## 5.2 Existing Compression Techniques

Modern compact networks [216, 241, 281] aim to reduce the "size" of the model, i.e., the number of model parameters that must be stored in, and retrieved from, memory. Similarly, techniques like weight pruning [84, 92, 107, 113, 280] and weight quantization [32, 48, 61, 62, 95, 164, 184, 207, 253], try to minimize the memory footprint used to store the weights of a trained model, either by reducing the effective size or omitting some (often ∼90%) of the parameters altogether.

As the weight footprints of such DNNs have shrunk, however, the relative footprint of their activations has increased to match or even exceed that of the weights even without pruning. Figure 5.2 illustrates this: while older (and larger) networks like VGG16-BN [230], ResNet18 [112], and WideResNet50 [273] have 5×–10× more weights than activations, modern compact networks like MnasNet1_0 [241], ShuffleNet v2 [281], and MobileNet v2 [216] have as many activations as weights (or more) even when processing a single input (i.e., no batching).

### 5.2.1 Drawbacks of Existing Techniques

Recalling from chapter 2, techniques like quantization and compression are more difficult to apply to activations than weights. Because activations change for every input at inference time while weights stay constant once the model is trained, activations cannot be quantized off-line; as a result, the model cannot be retrained to reverse the accuracy-drop due to activation quantization, as is commonly done with weight quantization [61, 106].

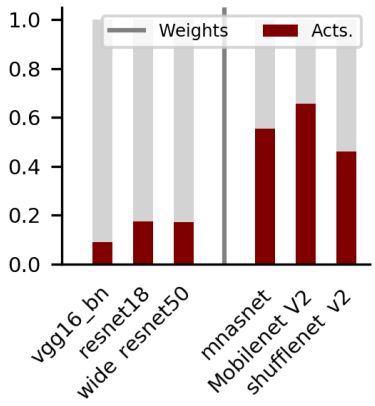



Figure 5.2: Share of the activation and weight values. The right side shows older models where weights can be up to an order of magnitude more than activations. On the other hand, the more modern and compact networks are illustrated on the left side and contain almost equal number of weights and activation values.

Methods like quantization-aware training [61, 95] and fine-tuning [106] try to mitigate the accuracy drop due to activation quantization during the training phase, by calibrating quantization ranges on different inputs requiring access to some training data. Often it is impossible with commercial vendors on account of security, privacy, or trade secret concerns to access the training data. Because of this, these techniques cannot be used post-deployment. Many of post-training quantization (PTQ methods attempt to quantize the activations without retraining or fine-tuning [32, 48, 62, 164, 184, 207, 253], while other proposals fine-tune the model using synthetic data that is distilled from the statistics stored in batch normalization layers [48].

While these PTQ techniques are effective in quantizing activations down to 8 bits, they fail dramatically at lower bit-widths. Figure 5.3 demonstrates this accuracy drop where weights are kept at 8-bit precision, but activation values are quantized to 6/5/4-bit precision for the state-of-the-art technique, ZeroQ [48]. When all activation layers are quantized, the accuracy of MnasNet1_0 using 4-bit activations drops to 0.12%. In contrast, the Channeleon technique we propose in this paper maintains a top-1 accuracy at 62.2%.

Apart from the accuracy loss, a key drawback of these state-of-the-art PTQ techniques is that they rely on per-channel statistics from the batch normalization layers, and require these to be stored in a full-precision format.

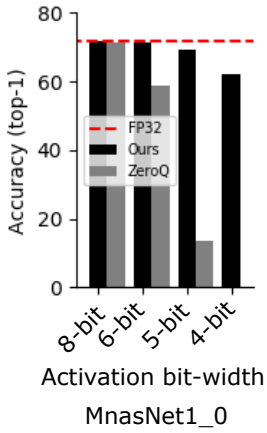



Figure 5.3: Impact of our proposal on the top-1 inference accuracy of the MnasNet1_0 networks compared to the best state-of-the-art activation value quantization methods [48].

This means that these methods fail on "Normalization Free" networks [41], which eschew batch normalization. Finally, batch normalization layers in off-the-shelf networks are often folded into the preceding layers for computation efficiency, and thereby lose the distinction between the weight values and normalization constants.

### 5.2.2 Channeleon Key Insights

The approach we propose in this chapter is based on two key observations:

1. Activation distributions are similar across large *groups* of channels, and that channels in each such group can be compressed together.
2. At very low bit-widths (less than 8 bits), the non-uniformity of activation distributions must be taken into account during quantization and compression.

Compressing each channel group together avoids the overheads of compressing each channel separately — especially prominent towards the end layers of deep convolutional networks — as well as the accuracy losses due to using the same quantization parameters for all of a layer's activations. Further, while traditional non-linear quantization techniques are too computationally expensive for inference, we develop a variant that incurs minimal computation overheads.

Table 5.1: Accuracy (top-1) improvement of the per-channel quantization over per-layer quantization. Per-channel quantization matches the 6-bit budget method match (or even outperform) the accuracy of the 8-bit per-layer quantization method. All weights are quantized to 8-bit.

| Method | A bits | **MnasNet1_0** | **MobileNet v2** | **ShuffleNet v2** |
|---|---|---|---|---|
| Per-Layer | 8 | 71.300 | 68.898 | 65.200 |
| | 6 | 58.812 | 38.438 | 56.490 |
| Per-Channel | 6 | 71.210 | 69.896 | 65.168 |

## 5.3 The Opportunity for Channel Clustering

### 5.3.1 Compressing Activation Tensors

Below, we first discuss two approaches to perform compression — compression based on computing the statistics of the entire layer, and compression based on computing the statistics for every channel separately — and their weaknesses, and then develop the proposed channel grouping technique in the next section.

**Per-layer compression.** The most common [32, 48, 62, 164, 184, 207, 253] way to compress the activation values is to quantize and compress an entire activation layer all at once based on statistics collected during calibration. In this approach, the quantization statistics — such as the min/max thresholds and the scale — are shared across all activations in a layer.

While this approach is simple and induces minimal overheads — the compression metadata are stored only once per layer — it also wastes compression opportunities.

Figure 5.4(a)(1) shows the distribution of the entire activation tensor after the first CONV layer of ShuffleNet v2 (including batch-norm ReLU) for two input samples. Quantizing the entire activation tensor uniformly will yield min/max thresholds of 0 and 6. However, many of the channels, shown in Figure 5.4(a)(2–7), have much narrower activation value ranges, and only use a small portion of the $[0, 6]$ range. This leads to overquantization, and substantially impacts accuracy as compression factors grow: Table 5.1 shows that while per-layer quantization works acceptably when targeting 8-bit activations, it suffers significant accuracy loss when targeting 6-bit activations.

**Per-channel compression.** An alternative to compressing the entire activation layer at once is to consider each channel independently [32]. This

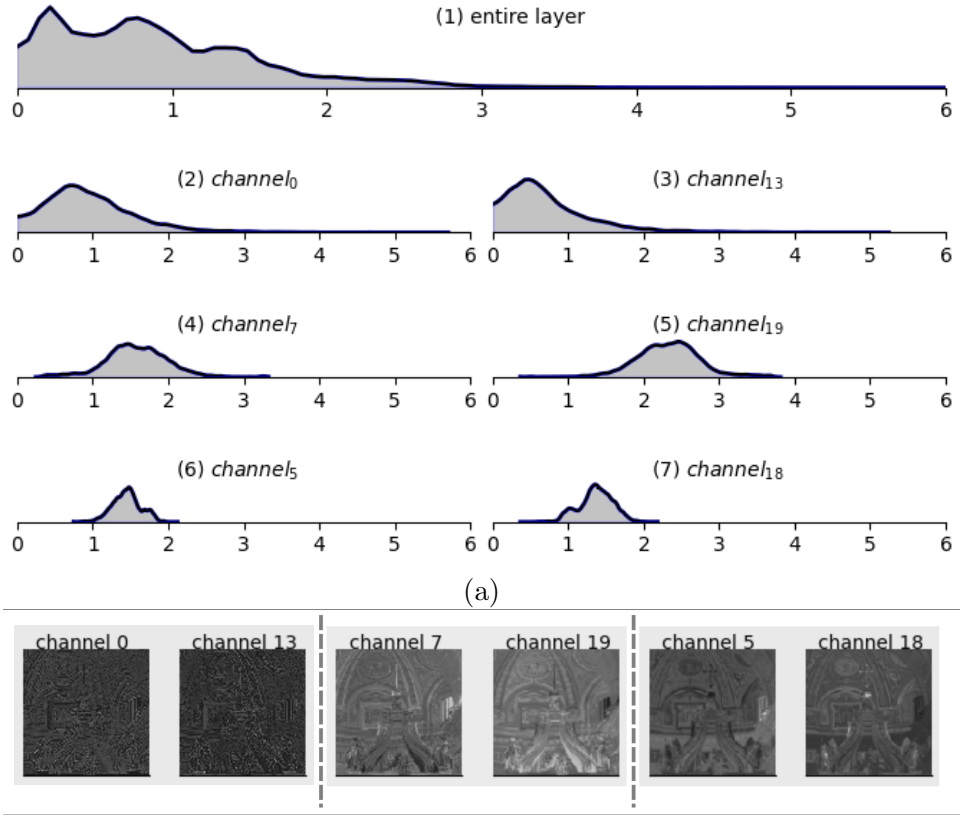



Figure 5.4: (a) Distribution of the activation values in the first layer of the ShuffleNet v2 illustrated on 2 input images (blue and red colors). The distribution of the entire values in the layer tensor is shown on the top; the distribution of values for 6 of the channels also separately illustrated on the bottom. The distributions and their statistics $(\mu, \sigma)$ vary for every channel, and are different from the distribution of the entire tensor. (b) Visualizing the same 6 channels as images for the first input image; while each image has unique features, there is similarity among pairs in each row.

allows the quantization to account for the different activation distributions present in each channel (Figure 5.4), and maintains accuracy at lower bit-widths (Table 5.1).

However, quantizing each channel independently also means an increased storage cost for the quantization metadata, which can significantly reduce the effective amount of compression. For example, consider the last three blocks of MobileNet v2, where activations are $7 \times 7$. Moving from 8-bit quantization to 6-bit quantization theoretically saves two bits per activation, but with the overhead of storing the quantization metadata for each channel, the savings reduce to only 0.7 bits per activation. Because of this, per-channel quantization is also undesirable as activations are quantized to narrower bit-widths.

## 5.4 Quantization Methods

Quantization is an effective technique to reduce bitwidth [33, 81, 94, 131, 146]. A quantization function *Quant* maps the set of reals $\mathcal{R}$ (full-precision data) to $Q_b$ (low-precision data at bit-width $b$) with a bitwidth of $b$. *Quant* is chosen to minimize the *quantization error* between the full-precision values and their low-precision representations, usually measured as the mean squared error (MSE).

Broadly, there are two types of quantization methods: uniform and non-uniform.

### 5.4.1 Uniform Quantization

A uniform quantizer linearly maps the full-precision data ($x$) into low-precision data ($\hat{x}$) with a scaling factor (denoted as *scale*). Uniform symmetric quantization ($Q^{\mathcal{U},sym}$) maps full-precision data values as: $Q_b^{\mathcal{U},sym} = scale \times \{-2^{b-1}, \ldots 0, \ldots, +2^{b-1} - 1\}$, where $b$ is the bitwidth of the low-precision data. Because most activations in a layer generally fall into a limited range, the quantization is usually cut off at upper and lower thresholds $\mathrm{X}_{max}$ and $\mathrm{X}_{min}$, i.e., the maximum and minimum values.

Because the commonplace ReLU activations set all negative values to 0, activations often use asymmetrical quantizers ($Q^{\mathcal{U},asym}$) which map the values as: $Q_b^{\mathcal{U},asym} = offset + scale \times \{0, ..., +2^b - 1\}$, where the values are shifted (denoted by *shift*) appropriately.

The quantized integer value $x_q$ and low-precision value $\hat{x}$ are then defined as:

$$
\begin{aligned}
shift &= X_{min}, \quad step = (X_{max} - X_{min})/2^b \\
x_q &= \lfloor \frac{x - shift}{step} \rceil \\
\hat{x} &= shift + step \times x_q + \frac{step}{2}
\end{aligned}
\tag{5.1}
$$

While uniform quantization can be performed with low latency overheads, it treats all data values equally and does not adapt to the the data distribution. For instance, using a uniform quantization for data that follows (say) a log-normal distribution will increase the quantization error and can cause a drop in accuracy.

### 5.4.2 Non-Uniform Quantization

A non-uniform quantizer non-linearly maps the full precision data ($x$) into low-precision data ($\hat{x}$). While this increases complexity, a non-uniform quantizer tends to have a smaller drop in accuracy as its quantization error is low.

In non-uniform quantization, the quantized values no longer contain a particular approximate value, but an index into a table of values. Finding the intervals that should be quantized as one value is often accomplished through clustering techniques (using, e.g., K-Means), with the centroids representing the quantized values and the cluster boundaries representing the quantization regions [106, 201].

Figure 5.5(b) shows how such a method chooses centroids $cent_0, \ldots, cent_{K-1}$. The boundaries of the quantization regions $[r_0, r_1), \ldots, [r_{K-1}, r_K)$ can be expressed as the mean of the centroid values. In iterative quantization (e.g., based on K-Means), these steps are repeated iteratively until the MSE for the quantized values is less than the desired threshold. The quantized integer value $x_q$ and low-precision value $\hat{x}$, is then defined as:

$$
\begin{aligned}
\hat{x} &= cent_i = \mathbb{E}[X | r_i \leqslant x < r_{i+1}] \\
r_i &= \begin{cases} X_{max}, & \text{if } i = K \\ X_{min}, & \text{if } i = 0 \\ (cent_i + cent_{i-1})/2, & \text{otherwise} \end{cases} \\
x_q &= i \text{ s.t. } r_i \leqslant x < r_{i+1},
\end{aligned}
\tag{5.2}
$$

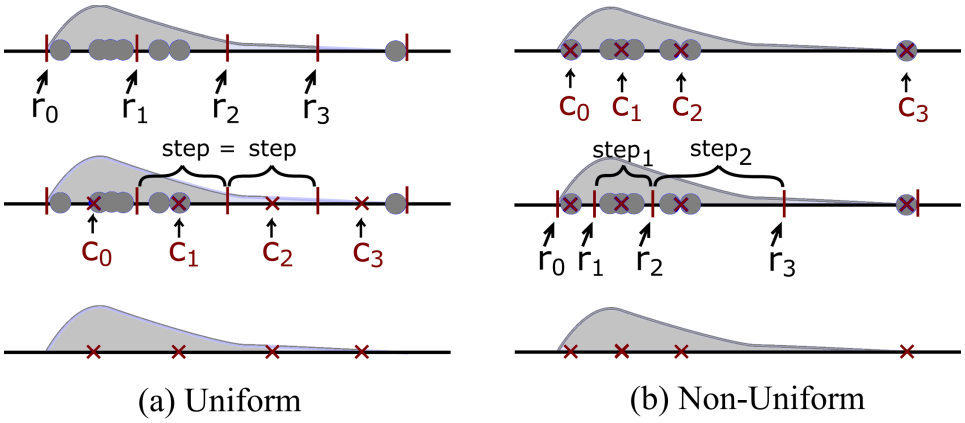



Figure 5.5: Quantizing values of a non-uniform distribution (blue dots) with a 2-bit budget. (a) uniform quantization: first, given the min/max of the values, the entire range will be divided into 4 equally spaced regions, then middle point of each region (i.e., the red cross $c_0$ ... $c_3$ ) will be chosen as the approximate values of each region. (b) Non-uniform: Assuming an initial region, the centroids are calculated for each region, and then the region boundaries $r_1$,... $r_3$ will be re-calculated accordingly to the mean of centroids. This process can happen iteratively to have less quantization error. Finally, the centroids can be used as the approximate value of values in each corresponding region. The approximated values using non-uniform quantization better match the blue dots than the uniform method.

## 5.5 Dynamic Clustering

Channeleon quantizes activations by combining two techniques: (i) it groups channels whose activations exhibit similarity and compresses them together, and (ii) it uses a Gaussian mixture approach to approximate and efficiently represent activation values. We discuss each of these in turn, and then present the quantization as part of the complete inference algorithm.

### 5.5.1 Dynamic Channel Grouping

To compress activation tensors, Channeleon dynamically groups subsets of channels with similar activation statistics, and uses the common statistics to compress those channels together.

**Channel-Group quantization.** To develop a better quantizations, we observe that, while activations are distributed differently in different channels, there are groups of channels with similar distributions. This is visible in the activation distributions in Figure 5.4(a): in each row, the distributions are visually similar. Figure 5.4(b) shows the same pairs of channels as 2D image; the similarity is also visible.

This observation suggests a best-of-both-worlds approach to activation quantization: instead of treating all activations in a layer as one distribution or treating all channels separately, detect similarity among channel activation distributions.

If an activation tensor has $C$ channels and they can be arranged in $K$ groups, we only need to store one set of quantization parameters for each of the $K$ groups, not for each of the $C$ channels.

The remaining question is how to detect channel similarity. We empirically found that collecting $\langle \sigma, min, max \rangle$ statistics for each channel and clustering them using K-Means allowed us to detect channels that could share quantization parameters without impacting inference accuracy.

Table 5.2 shows that applying this method gives an effective $K$ that is much smaller than $C$, resulting in significant savings: for example, for MnasNet1_0, channel grouping can reduce the total number of quantizers by more than 22× (from $18.9K \rightarrow 0.8K$) while maintaining top-1 accuracy.

### 5.5.2 Non-Uniform Quantization

Figure 5.4(a) also shows that — especially if channels are considered separately — activations are not uniformly distributed between the cutoff thresholds. Quantizing these ranges uniformly is acceptable if the range is suffi-

Table 5.2: Accuracy (top-1) comparison of the group-channel quantization and per-channel quantization. Grouping channels into 16 groups reduces the number of quantizers by an order of magnitude while preserving the final model accuracy with 8-bit weights.

| Model | **Per-Channel** (6-bit A) | | **Per-Group** (6-bit A) | |
|---|---|---|---|---|
| | Q. unit | top-1 | Q. unit | top-1 |
| **MnasNet1_0** | 18,961 | 71.210 | 833 (22.8×) | 71.272 |
| **MobileNet v2** | 17,851 | 69.896 | 995 (17.9×) | 69.904 |
| **ShuffleNet v2** | 8,115 | 65.168 | 913 (8.9×) | 65.164 |

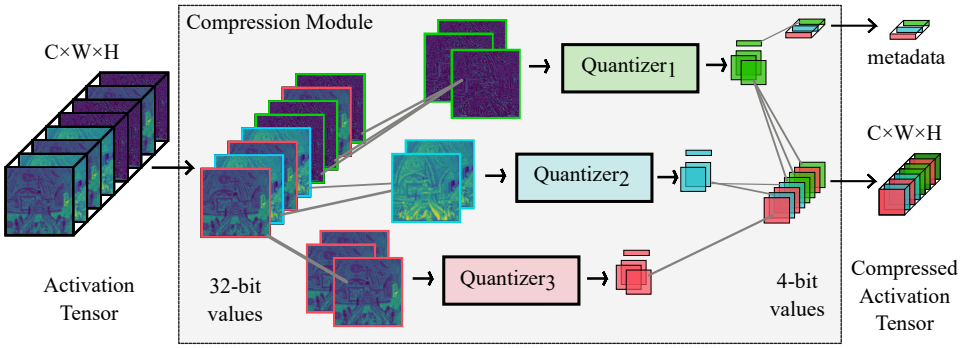



Figure 5.6: Illustration of the channel clustering method in Channeleon on tensor With $C = 8$ channels, each with the dimension of $W \times H$, and $K = 3$ number of channel clusters. First, the similar channels will be grouped together, then each group will be quantized with their corresponding quantizer. The output will be the the integer indices for each 8 channels with the same $W \times H$ dimension (with lower bit-width per value) as well as the metadata for each of the 3 groups.

ciently fine-grained — say 8-bit quantization — but quickly fails as fewer bits are used for the quantization: Table 5.3 shows that uniform quantization causes significant accuracy losses at 4-bit quantizations.

One possibility is to use clustering-based non-linear quantization, as has been proposed in the context of weight quantization [106]. However, there is a key difference: while weight quantization can occur off-line *before* deployment, activation quantization must be done on-line *during* inference. And clustering is computationally expensive: using three iterations of K-Means to quantize the activations of MobileNet v2 requires 400M operations, compared to the 287M operations required for the convolution operations.

Channeleon therefore adapts clustering-based quantization method to the inference-time performance constraints by sampling and clustering a subset of the activations. We sample about 1% of the activations from each channel group as they are produced, and use them to determine the quantization parameters. This produces computation overheads in the range of ∼1%, well below the overhead needed for the actual quantization (cf. Figure 5.7).

Table 5.3: top-1 accuracy comparison of uniform quantization and non-uniform quantization under the same group clustering setting (16 channel groups). Accuracy is significantly improved at low activation bit-widths.

| **Quant.** | **MnasNet1_0** | | |
|---|---|---|---|
| **method** | 6-bit | 5-bit | 4-bit |
| **Uniform** | 71.272 | 68.628 | 57.328 |
| **Non-Unif.** | 71.436 (+0.164) | 69.179 (+0.551) | 62.228 (+4.9) |

As the activation values are sparse after the ReLU layer, we refine the compression by first encoding zero values with a bitmask (1 bit per value) and then applying K-Means on the non-zero values.

### 5.5.3 Channeleon in the Inference Process

After the computation for each layer is complete (including batch-norm and activation function), the output activation tensor $T^{(i)}$ is collected and quantized.

This proceeds by first clustering the channels of $T^{(i)}$ into $K_l$ clusters (a hyperparameter selected independently for each layer to balance accuracy and memory footprint). This process assigns a group ID $g\_id$ to each channel.

Next, each channel group is quantized separately by sampling ∼1% of the activations and clustering them into $K_g$ clusters using K-Means ($K_g$ = 16 for 4-bit quantization, 32 for 5-bit quantization, and so on). This results in the quantized activations $qt$ for the channel group and the quantization metadata (cluster centroids), which are collected in $qT$ and $M_l$, respectively.

Finally, the quantized output activation tensor $qT^{(i)}$ is used as input activations for the next layer.

## 5.6 Methodology

We evaluate the benefits of Channeleon on three modern compact DNNs, MnasNet1_0 [241], MobileNet v2 [216], and ShuffleNet v2 [281] on the ImageNet [209] dataset.
We implement Channeleon in 'Distiller' [287] using the PyTorch [197] framework. We quantize all the Convolutional (CONV) and Fully Connected (FC ) layers, including the first and the last layers. As it is common in inference, the Batch Normalization values are folded into the weights of corresponding layers. Furthermore, ReLU layers are also fused with corresponding previous layers. To highlight the benefits of compressing activations, we fix the weights bit-width to 8-bit in all our experiments.

To quantify the impact of Channeleon on compressing non-zero activations, we encode the zero activations with a bitmask and only quantize the non-zero activations. Channeleon uses K-Means clustering with linear initialization to perform the quantization task. We observed that sampling only 1% of the activations enables us to capture the characteristics of all the activations in a group. Sampling reduces the computational overheads of K-Means clustering significantly. This is even more beneficial when we have multiple channels, especially at the later DNN layers, where the average number of members per cluster is observed to be significantly higher than the earlier layers.

We observed that on average, K-Means clustering for a group requires up to three iterations before ending the clustering task. We also tried various $K$ values and found $K = 16$ forms good clusters while keeping a low overhead.

Table 5.4: Accuracy improvement from channel clustering in Channeleon over a naive layer-wise quantization. Channel clustering in Channeleon helps significantly even at low (5-bit and 4-bit) activation bit-widths.

| Act bits | Model | Layer wise | Channeleon (ours) | top-1 diff. |
|---|---|---|---|---|
| **5-bit** | MnasNet1_0 | 63.74 | 69.18 | **+5.44** |
| | MobileNet v2 | 64.61 | 67.45 | **+2.84** |
| | ShuffleNet v2 | 57.80 | 64.57 | **+6.77** |
| **4-bit** | MnasNet1_0 | 48.85 | 62.23 | **+13.38** |
| | MobileNet v2 | 47.60 | 59.00 | **+11.40** |
| | ShuffleNet v2 | 39.92 | 60.12 | **+20.20** |

## 5.7 Evaluation Results

### 5.7.1 Ablation Study

We perform an ablation study for the two components of Channeleon, namely (a) Channel Grouping, and (b) Non-Uniform Quantization.

**Impact of Channel Grouping**
We conduct experiments on MobileNet v2 and ShuffleNet v2 and MnasNet1_0 with 8-bit weight quantization with 5-bit and 4-bit activation quantization. Table 5.4 shows that Channel Grouping quantization outperforms the accuracy of layer-wise quantization by at least 2.84% and up to 20.2% points. This showcases that channel grouping in Channeleon benefits over layer-wise quantization even for low activation bit-widths.

**Impact of Non-Uniform Quantization**
To validate the effectiveness of non-uniform quantization in Channeleon, we use channel clustering methods on a fixed number of channel groups. However, for each channel group, we perform non-linear quantization and compare these against uniform quantization. Table 5.5 shows the effect of non-uniform quantization with 5-bit and 4-bit activations. Channeleon consistently outperforms the group-wise uniform quantization across all networks, and achieves up to 7.78% top-1 improvement (4.9% – 7.78%) for 4-bit activations.

Table 5.5: Accuracy improvement from non-uniform quantization in Channeleon over widely used uniform quantization at a channel-group granularity. Non-uniform clustering in Channeleon helps significantly even at low (5-bit and 4-bit) activation bit-widths.

| **Act bits** | **Model** | **Groupwise Uniform** | **Channeleon** (ours) | **top-1 diff.** |
|---|---|---|---|---|
| **5-bit** | MnasNet1_0 | 68.63 | 69.18 | +0.55 |
| | MobileNet v2 | 66.72 | 67.45 | +0.73 |
| | ShuffleNet v2 | 63.14 | 64.57 | +1.43 |
| **4-bit** | MnasNet1_0 | 57.33 | 62.23 | +4.9 |
| | MobileNet v2 | 52.25 | 59.00 | +6.75 |
| | ShuffleNet v2 | 52.34 | 60.12 | +7.78 |

### 5.7.2 Classification Results

Table 5.6 reports the accuracy of Channeleon compared to the state-of-the-art 'datafree' PTQ method called ZeroQ [48]. We also report the results for an idealized version of ZeroQ, denoted by ZeroQ+, which uses a part of the training dataset for calibration. This establishes the upper bound for the distillation mechanism in ZeroQ. Similar to ZeroQ, our baseline quantizes weights into 8-bit integers and keeps the activation values at 32-bit floating point. The baseline accuracies are reported with *. We also report the accuracy of a full-precision model in parenthesis below each model. The accuracy gap between the baseline and the full-precision model is due to weight quantization and can be mitigated by orthogonal weight quantization techniques.

Our experiments show that using 8-bit activation quantization can result in close-to-baseline accuracies for the ZeroQ+ techinique. However, reducing the activation bitwidth to 6-bit (and below) has a significant impact on the accuracy for all models. For instance, even with ZeroQ+, MobileNet v2 shows an accuracy drop from 68.9% to 38.44% when we reduce the activation bitwidth from 8-bits to 6-bits. On the other hand, Channeleon consistently outperforms ZeroQ and ZeroQ+ on all models at reduced bit-widths. After using Channeleon, MobileNet v2 increases the accuracy to 69.35% even while using 6-bit activation values. Furthermore, the 6-bit activation version of Channeleon produces higher accuracies than the 8-bit ZeroQ+, and are within **0.21% to 0.49% of the baseline accuracy**.

The benefits from Channeleon start becoming more obvious at lower bit-widths. On a setup with 4-bit activation values, the prior work including

Table 5.6: Accuracy comparison of Channeleon on quantized post-training models. The accuracy of the unquantized network is reported in the parenthesis below each model. The baseline is the model with INT8 weights and unquantized 32-bit floating point activations. ZeroQ is our implementation of [48] with batchnorm folding and ReLU fusion. ZeroQ+ denotes the idealized version of ZeroQ when it has the ideal calibration dataset (subset of the training dataset). Even idealized ZeroQ suffers on accuracy on lower activation bit-widths while Channeleon consistently outperforms it at each bit-width.

| | **Method** | top-1 Accuracy | | | |
|---|---|---|---|---|---|
| **MnasNet1_0** (73.46) | **baseline** | 71.93* | | | |
| | | 8-bit | 6-bit | 5-bit | 4-bit |
| | **ZeroQ** | 12.13 | 10.23 | 5.10 | 0.75 |
| | **ZeroQ+** | 71.30 | 58.81 | 13.64 | 0.18 |
| | **Channeleon** | | **71.44** | **69.18** | **62.23** |
| **MobileNet v2** (71.89) | **baseline** | 70.39* | | | |
| | | 8-bit | 6-bit | 5-bit | 4-bit |
| | **ZeroQ** | 22.34 | 15.15 | 3.30 | 0.22 |
| | **ZeroQ+** | 68.90 | 38.44 | 2.64 | 0.13 |
| | **Channeleon** | | **69.35** | **67.45** | **59.00** |
| **ShuffleNet v2** (69.36) | **baseline** | 65.50* | | | |
| | | 8-bit | 6-bit | 5-bit | 4-bit |
| | **ZeroQ** | 15.60 | 14.77 | 8.36 | 2.98 |
| | **ZeroQ+** | 65.20 | 56.49 | 16.81 | 0.26 |
| | **Channeleon** | | **65.29** | **64.57** | **60.12** |

Table 5.7: Activation storage requirements of a baseline already quantized to 8 bits, and Channeleon at 6, 5, and 4 bit widths. The amount of overhead of storing centriods and bitmasks are included below Channeleon activation footprint. Channel index overhead is negligible and not included here. Last column shows ZeroQ's total activation size using 4 bits (per tensor scale and shift parameters are negligible in size and ommited here). As ZeroQ uses linear quantization only, the activation size of 8-bit ZeroQ is equivalent to values reported in the 8-bit baseline.

| **Model** | **baseline** | **Channeleon** | | | **ZeroQ** |
|---|---|---|---|---|---|
| | 8-bit | 6-bit | 5-bit | 4-bit | 4-bit |
| **MnasNet1_0** (20.82MB) | **5.21MB** | **3.28MB** 6%/20% | **2.78MB** 4%/23% | **2.33MB** 2%/28% | **2.61MB** |
| **MobileNet v2** (25.48MB) | **6.37MB** | **4.22MB** 5%/19% | **3.58MB** 3%/22% | **3.00MB** 2%/26% | **3.2MB** |
| **ShuffleNet v2** (7.44MB) | **1.86MB** | **1.42MB** 15%/16% | **1.15MB** 10%/20% | **0.94MB** 6%/25% | **0.94MB** |

the idealized version ZeroQ+ has an accuracy of under 3% while Channeleon maintains an accuracy of at least 59%. As an example, in the case of ShuffleNet v2, Channeleon causes only a 5.38% reduction of baseline accuracy while ZeroQ+ drops the baseline accuracy by a massive 65.24% points.

### 5.7.3 Memory footprint analysis

Table 5.7 reports the total activation size of each layer before and after the compression. The total number of activation in the entire network is showcased in parenthesis below each network. We also report the overhead of the metadata in the format of '(centriods/bitmasks)' in terms of the percentage of the total footprint. For example, the activations of the MnasNet1_0, can be compressed from 5.21MB to 2.33MB. This is 2.24× lower as Channeleon reduces the bitwidth from 8-bits in the baseline to 4-bits, including the 2%(0.05MB) centroid and 28%(0.65MB) bitmask overheads.

The channel index that identifies the channel group incurs negligible at overheads ranging from 0.2% to 0.4% of the compressed size, and so we omit that in the table. This reduced memory footprint is crucial for energy-efficient inferences as Dynamic RAM accesses are order of magnitude expensive than arithmetic operations [105].

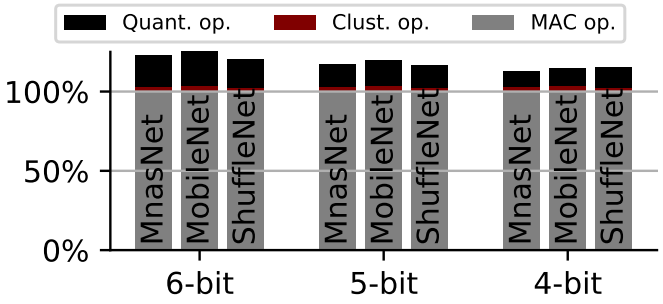



Figure 5.7: The overhead of Channeleon normalized to the network inference operation. Note that the gray bar shows the MAC operation in inference, while the black and red bars, shows the overhead in terms of the simple subtraction operations.

**Cost analysis**

To evaluate the overhead of Channeleon, we profiled the K-Means iterations and sampled activation sizes etc during runtime. We calculate average operations performed by K-Means relative to network inference operations. As shown in Figure 5.7, channel clustering adds only 2% to 3.5% more operations as compared with total inference operations. Furthermore, non-linear quantization only adds an additional 10% to 22% as compared with total inference operations. Moreover, the dominant primitive operations in K-Means are subtraction whereas the more expensive multiply-and-accumulate (MAC) operations dominate during network inference.

The energy consumption is more than than 4× less for operations required for clustering compared with operations at network inference (based on [105]). Therefore, we project that Channeleon only adds 3% to 6.4% energy overheads. As DRAM accesses are an order of magnitude more energy consuming than arithmetic operations [105], this overhead is negligible comparing to the saving obtained from the reduced memory footprint.

## 5.8 Summary

In this chapter, we tackle the problem of activation map compression in DNNs. We observe that modern compact DNNs, unlike older networks, tend to have almost equal proportions of weights and activations. Thus traditional techniques that statically compress weights during training are not feasible for activations (as they are dynamic) on these in-deployment devices.

The key observation here is that many activation channels share similar statistics and this makes them clusterable. Therefore, we propose Channeleon, a computationally efficient technique that enables activation quantization by using sampling and K-Means clustering across channels. Moreover, Channeleon does not require access to any training data or model hyperparameters.

# Chapter 6

# Sparse Training Accelerator

In this chapter[3], we tackle the problem of accelerating a sparse training from scratch. First, we review an sparse training algorithm [92] where the weight values are categorized as more important and less important over the course of training. This is done by considering the entire set of gradient values and keeping the ones with the highest change rather than looking at the gradients or weights individually. Then, we show how to adapt this algorithm to make it suitable for hardware accelerator implementation; the adapted algorithm achieves 3.9×–11.7× sparsity while maintaining unpruned accuracy on tasks like CIFAR10 and ImageNet.

We then propose a hardware architecture that adapts a standard 2D-PE-array inference accelerator to enable sparse training without incurring the dataflow limitations and interconnect complexity of the only prior sparse training accelerator proposal [278] and achieves much higher sparsity.

Finally, we develop a sparse data representation suitable for training access patterns, and an inexpensive load-balancing technique that preserves maximum spatio-temporal reuse without complicating the on-chip interconnect. Most of the modifications are not specific to the sparse training method we adapt, but rather are necessary for accelerating any existing sparse training approach.

Compared to an equivalent accelerator that does not support training-time sparsity, Procrustes uses 2.27×–3.26× less energy and offers 2.28×–4× speedup without compromising accuracy on state-of-the art networks on ImageNet and CIFAR-10.

## 6.1 DNN training

Stochastic gradient descent (SGD) — the de facto standard training algorithm for deep neural networks [154] — comprises three stages, illustrated

[3]Parts of this chapter appear as: D. Yang, A. Ghasemazar, X. Ren, M. Golub, G. Lemieux, M. Lis, "Procrustes: a Dataflow and Accelerator for Sparse Deep Neural Network Training", MICRO, 2020.

in Figure 6.2:

1. The *forward pass* runs the inference algorithm to determine the model's predictions for training inputs and calculate the loss $\mathcal{L}$ (i.e., the training error). For a convolutional layer, this consists of convolving the input activation (iact) tensor $x$ with a set of filters $w$ to obtain the output activation (oact) tensor $y$ (Figure 6.2a).

2. The *backward pass* back-propagates the loss gradient across the model's layers. For a convolutional layer, this is done by convolving the loss gradient with respect to the oacts $\frac{\partial \mathcal{L}}{\partial y}$ with filters $w$; unlike in the forward pass, however, each filter is first rotated 180$\circ$ (Figure 6.2b).

3. The *weight update pass* determines how much a weight $w$ should be adjusted to decrease the loss by computing the gradient $\frac{\partial \mathcal{L}}{\partial w}$. For a convolutional layer, this consists of convolving the backpropagated loss gradient with respect to the oacts $\frac{\partial \mathcal{L}}{\partial y}$ with the input activations (iacts) $x$ (Figure 6.2c).

In fully connected layers, $x$ and $y$ are 1D vectors, a weight matrix replaces the weight filters, inner product replaces convolution, and matrix transpose replaces the 180$\circ$ rotation.

## 6.2 Potential Savings of Sparse-From-Scratch

Pruning techniques can typically reduce the weight count by an order of magnitude [106, 107, 109, 153, 159, 166, 261, etc.]. This sparsity comes at the cost of irregular memory accesses and computation patterns, and several accelerators have been proposed to enable efficient inference on sparse models [58, 93, 105, 193, 279, etc.].

None of these approaches were, however, designed for energy-efficient *training*. This is because they target a context where pruning occurs *after* training: a model is first trained with the full parameter set, then pruned, and finally re-trained to recover accuracy [107]. While this saves energy at inference time, training the pruned network takes *more* time and energy than training an equivalent dense network to the same accuracy. Skipping the pre-training step is not an option: even if oracular knowledge of the pruned model connectivity is assumed, training the pruned model from scratch sacrifices accuracy compared to the original network [107, 159].

Still, the very existence of pruned networks suggests that it must be possible to somehow train them. Recent work has demonstrated that a model

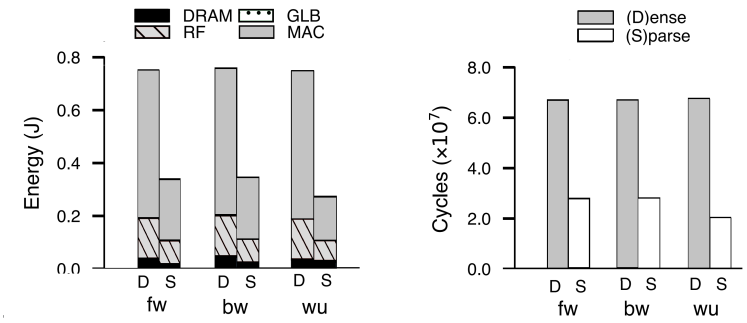
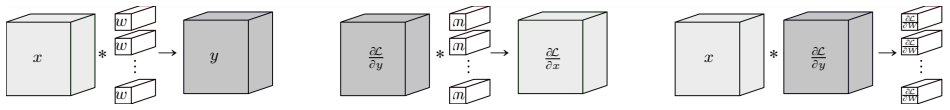



Figure 6.1: Potential training energy savings and speedup from ideally leveraging all weight sparsity (here, 5×) while training VGG-S (15M weights) to convergence with Dropback [92]. fw/bw/wu = forward/backward/weight-update phases.

pruned by an order of magnitude can be trained *provided* that the initialization for the unpruned subset of weights is preserved [84]; this can be achieved either by dynamically selecting the most productive gradient subspace [92] or by iteratively increasing sparsity [179, 278].

Ideally, such sparse-from-scratch training can offer significant savings. Figure 6.1 shows this for VGG-S [272] pruned 5× (15M→3M weights) using the Dropback algorithm [92], in an idealized 16×16 PEs training system where (i) sparsity is evenly distributed within each layer so all PEs receive the same workload (i.e., perfect load balancing), and (ii) sparse weights are stored in an idealized compressed format with no overhead, and (iii) retained weights selection is instant and cost-free (see Figure 4.7.4 for setup details). While the exact improvement varies with the geometry and sparsity of each layer, leveraging 5× sparsity can yield up to 2.6× speedup with 2.3× less energy consumption over the entire network.

In practice, however, none of the existing sparse training methods can reach this potential. Most [84, 92, 278] require sorting all weights to determine the parameters to retain; with weight counts in the tens of millions, sorting is an expensive proposition. Several [179, 278] achieve only small pruning factors and suffer accuracy loss. Some [84, 278] prune the model very gradually; this implies (i) no peak memory footprint reduction, (ii) mediocre energy savings because the average sparsity is low during most of the training process, and (iii) the need to support two weight storage formats (dense and sparse) and switch formats mid-way during training. The remaining technique [92] maintains the target weight sparsity throughout




(a) forward: $x * W \rightarrow y$ (b) backward: $\frac{\partial \mathcal{L}}{\partial y} * W^{\circlearrowleft} \rightarrow \frac{\partial \mathcal{L}}{\partial x}$ (c) weight update: $x * \frac{\partial \mathcal{L}}{\partial y} \rightarrow \frac{\partial \mathcal{L}}{\partial W}$

Figure 6.2: CNN training consists of (a) the forward pass, (b) the backward pass, and (c) the weight update pass; minibatch size adds a fourth dimension to the activations. Weights are accessed in different order during the forward and backward passes. Training FC layers is similar but uses multiplication instead of convolution and $W^\top$ instead of $W^{\circlearrowleft}$ in the backward pass. $\mathcal{L}$ = loss; $x$ = iacts; $y$ = oacts; $W$ = weights; $\circlearrowleft = 180\circ$ filter-wise rotation.

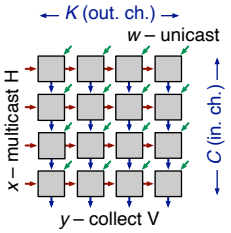
training, but gives up computation sparsity — a significant drawback for training, where weights are usually 32-bit floating-point numbers that are energetically expensive to multiply.

In addition, existing accelerators that support sparse inference are inadequate for sparse training. Weights are represented in formats that directly correspond to the dataflow being used [58, 93, 105, 193, 279, etc.]; this works well when weights are always accessed in the same order during inference, but does not support the different weight access patterns that arise in different phases of training (see section 6.1). Accelerators that perform load balancing (e.g., Sparten [93]) do this in software as a preprocessing step; this works for inference where weight sparsity is static, but not for training where weight sparsity changes dynamically. Finally, recent proposals like parashar2017scnn [193] and Sparten [93] use complex hardware to exploit two-sided sparsity (i.e., both weight and activation sparsity); this can be leveraged during the forward-pass phase of training, but usually does not exist in the backpropagation or weight update phases because the ubiquitous batch normalization destroys layer sparsity in the back-propagated gradient $\frac{\partial \mathcal{L}}{\partial y}$, so the additional hardware costs are not warranted for training. (We describe these challenges in more detail in chapter 2.)

## 6.3 Sparse Training Considerations

### 6.3.1 Sources of Sparsity

Inference accelerators that support sparsity [58, 93, 105, 193, 279, etc.] can leverage two sparsity sources: (a) zero-valued weights that result from pruning [107], and (b) zero-valued activations that result from the RELU acti-




| | fw | bw | wu |
|---|---|---|---|
| (H)orizontal | $x$ | $\partial\mathcal{L}/\partial x$ | $x$ |
| (V)ertical | $y$ | $\partial\mathcal{L}/\partial y$ | $\partial\mathcal{L}/\partial y$ |
| (U)nicast | $w$ | $w$ | $\partial\mathcal{L}/\partial w$ |

$w$ weights $x$ input activations
$y$ output activation partial sums

Figure 6.3: A weight-stationary mapping: input and output channel dimensions ($C$ and $K$) are distributed spatially.

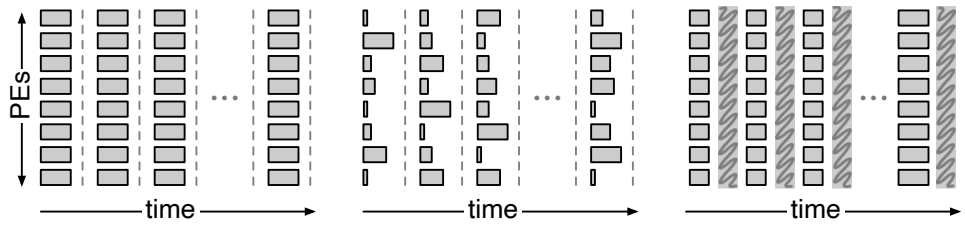
vation function [23]. With suitable hardware support, multiply-accumulate (MAC) operations that involve zero weights or activations can be skipped, while zero-valued weights and activations need not be stored if a suitable sparse data format is used; some accelerators can take advantage of both sparsity sources simultaneously [58, 93, 193].

During training, weight sparsity can also be used in the backward gradient propagation phase, and input activation sparsity in the weight update phase (cf. Figure 6.2). However, the back-propagated gradient $\frac{\partial\mathcal{L}}{\partial y}$ does not exhibit sparsity because of the prevalent use of batch normalization [128]: batch normalization layers are commonly used between CONV and RELU layers, which means that the $\frac{\partial\mathcal{L}}{\partial y}$ sparsity generated from backpropagating through RELU is destroyed by backpropagating through the batch normalization layer.

Designers of sparse training accelerators, therefore, are faced with a choice: either spend additional hardware to accelerate one third of the training process, or reduce hardware complexity but give up on leveraging activation sparsity in the forward pass. In this work, we focus on the latter approach.

### 6.3.2 Mappings, Dataflows, and Load Balancing

The computation required to evaluate a CONV layer can be represented as a seven-dimensional nested loop, where each loop traverses a different dimension of the operation space [192] (single-sample inference accelerators may not have the $N$ minibatch dimension). Regions of this operation space are then distributed as "work tiles" to different PEs by mapping two of the loops to the horizontal and vertical dimensions of a 2D PE array; together with exchanging the order in which loops are nested, this determines the




(a) dense (b) sparse +bcast (c) sparse –bcast

Figure 6.4: DNN computation on a 2D PE array with a weight-stationary $C, K$ mapping: (a) dense model, equal work and spatial reuse; (b) sparse model, unequal work but spatial iact/psum reuse; (c) sparse model, equal work but no spatial iact/psum reuse. □ = work tile; ▨ = overhead due to lack of reuse / complex interconnect. Each column is a full PE array's worth of work; a single layer's computation comprises many of these. ±bcast = with/without spatial weight reuse.

dataflow [149, 192].

Figure 6.3 shows the ubiquitous weight-stationary dataflow [24, 54, 139, 189, 226, 227, 238, 276, etc.], which results from mapping the $C, K$ dimensions across the PE array in the forward pass (mapping $R, S$ is less common due to small filter sizes); the corresponding mappings for the backward and weight-update passes are shown in the adjacent table.

In this mapping, each workload (e.g., DNN layer) is first divided into PE-sized work tiles, all of which have the same number of weights. The tiles are mapped among the PEs; once a PE receives one work-tile, the computation begins and runs until *all* work-tiles have finished. Finally, the next set of work-tiles is distributed among the PEs, and the process repeats until the entire layer has been evaluated (Figure 6.4a).

This mapping results in advantageous dataflow properties in a 2D PE array. Because all work tiles have the same amount of work, execution is naturally synchronized, and data can be spatially reused by broadcasting across multiple PEs. For example, in the forward pass in Figure 6.3, input activations are broadcast horizontally (read-only reuse), while partial sums are reduced vertically (read-write reuse). The dataflow patterns also allow the on-chip network to be simple: our example requires two one-dimensional flows (for the activations and the partial sums) and one unicast flow (for the weights); typically, those would be three separate interconnects.

However, difficulties arise when the network is sparse. At reasonable

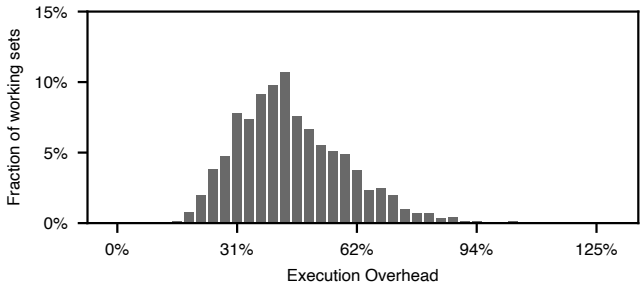



Figure 6.5: Load imbalance histogram of full-PE-array working sets (columns in Figure 6.4b) when training VGG-S [272]/CIFAR-10 [148] using Dropback sparse training [92]. A perfectly load-balanced workload would have 100% of the sets at 0% overhead.

pruning levels, on the order of 10% of the weights survive [107], with sparsity distributed unevenly among the worktiles (by chance and learning pressure). This leaves designers with two unpleasant alternatives:

1. Retain the same tiling and mapping of operations to the PE array as in the dense case. This preserves the single-dimensional dataflow patterns shown in Figure 6.3, allowing input activations and partial sums to be spatially reused. However, different amounts of work are distributed to different PEs, and utilization is low because latency is limited by the "slowest" PE (Figure 6.4b). Figure 6.5 shows how latency differs among full-PE-array sets of work tiles (i.e., columns in Figure 6.4b): frequently, the load imbalance causes execution time overheads in excess of 50%, and sometimes in excess of 100%.

2. Distribute an equal number of non-zero weights to each PEs. This balances the workload among the PEs (Figure 6.4b), but destroys the desirable single-dimensional on-chip traffic flow patterns of Figure 6.3 and severely reduces the benefits from spatial reuse. In addition, because related partial sums can be generated in any PE, a complex interconnect is required to reduce them [58, 278].

Choosing other dataflows also does not provide a panacea: for example, the activation-stationary dataflow used for some sparse accelerators [193] suffers similar issues in the weight update pass, requires two datatypes to be unicast, and suffers from low PE array utilization towards the tail of many networks where the activation tensors are small [58].

Subsection 6.5.3 describes how Procrustes employs the additional mini-batch dimension available during training to achieve effective load balancing while preserving a hardware-friendly dataflow and avoiding the need for a complex interconnect.

### 6.3.3 Sparse Weight Representation

Existing sparse-weight inference accelerators [58, 93, 105, 193, 279] employ a linear run-length encoding that is tightly coupled to the dataflow they use. For example, EIE [105] stores non-zero entries as an interleaved compressed sparse column (CSC) format, which permits a single column of an FC layer weight matrix $W$ to be streamed to the PE array to interact with the same input activations. This layout matches the dataflow during the forward pass, but makes it impossible to calculate addresses within a column of $W^\top$ in the backward pass.

Similarly, the compressed format used for CONV filters in SCNN [193] organizes filter layers so that all sparse filters with the same input channel (and different output channels) are adjacent. In the forward pass, this corresponds to SCNN's input-stationary dataflow where a single input activation is multiplied by all filters from the same input channel and the partial sums are distributed to different output channels; however, in the backward pass the equivalent gradient-stationary dataflow would need to compute addresses for all filters from one *output* channel, which is not possible due to varying filter sparsity.

Procrustes instead uses a variant compressed block format (CSB) [45] (subsection 6.5.2) to ensure that weights can be compressed but still read efficiently during all relevant training phases.

### 6.3.4 Sparse Training Algorithms

Training of sparse networks relies on the observation that dense deep neural networks contain small subnetworks (∼20% weights) that can be trained to match or exceed the original accuracy *provided* that the initial weight settings for the subnetwork are retained [84, 158]. In effect, most (∼80%–90%) of the weights serve as a scaffolding necessary only to identify the weights that should survive in the final pruned subnetwork.

Most of the proposed sparse training algorithms work by gradually increasing sparsity during the training process. The lottery ticket algorithm [84] prunes 20% of the network every 50,000 training iterations by removing the lowest-magnitude weights; the authors report 5–10× model size reduction

on CIFAR10 targets. Eager Pruning [278] follows a similar magnitude-based approach, but adds a feedback loop and a checkpoint-based rollback scheme to avoid overpruning; maintaining top-1 accuracy on ImageNet, it can prune ResNet50 2.4× (25.6M→10.8M weights) by removing 0.8% of the weights every 24,000 iterations. Both approaches rely on sorting all weight values to select which weights to keep.

Dynamic sparse reparametrization [179] starts by randomly distributing zero weights at the desired sparsity level, but allows the zeros to redistribute across the weight tensor during training. For ResNet50, for example, ∼200,000 additional parameters are set to zero every 1,000–8,000 iterations, but an equal number of weights are allowed to regrow after each pruning step. It avoids the need to sort all weights by using a value threshold adjusted via a set-point feedback loop whenever the network is pruned; however, the initial value of this threshold becomes a hyperparameter. ResNet50 can be pruned 3.5× (25.6M→7.3M) with some top-1 accuracy loss on ImageNet (−1.6%).

In contrast to the gradual pruning approaches [84, 179, 278], the Dropback algorithm [92] prunes the network from the beginning: only a fixed percentage of the parameters (e.g., 10%) are ever allowed to change. In every iteration, only the weights with the highest accumulated gradient survive (which again requires sorting), on the theory that this represents learning better than magnitude during early iterations; the pruned weights are reset to their initial values rather than to 0. Dropback prunes ResNet18 11.7× (11.7M→1M) while maintaining top-1 accuracy on ImageNet.

In this chapter, we focus on Dropback algorithm, which offers by far the highest compression ratios and introduces only one additional parameter (the sparsity factor) during training. Unfortunately, two aspects stand in the way of hardware acceleration: (a) pruned weights are not set to 0, and so MAC energy is not saved; and (b) millions of gradients must be sorted to determine which weights should be pruned. We demonstrate how to overcome these drawbacks and make Dropback algorithm hardware-friendly in section 6.4.

## 6.4 Sparse Training Algorithms in Hardware

To adapt Dropback algorithm to the requirements of an efficient hardware implementation, Procrustes

(i) creates computation sparsity by decaying initial weight values $W^{(0)}$ over the first 1,000 iterations, and

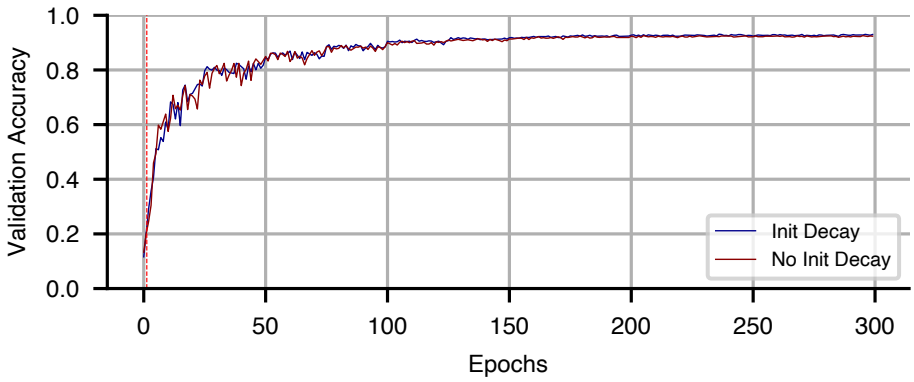



Figure 6.6: Validation accuracy over the course of training when initial weights decay 0.9× every iteration, compared to a baseline without decay (VGG-S on CIFAR-10). The dashed vertical line indicates the point at which all initial weights have decayed to zero (1,000 iterations, or early in the second epoch as epochs are 800 iterations each).

(ii) avoids the need to sort all gradients by using dynamic quantile estimation to continuously determine a threshold value that tracks the target sparsity.

We discuss the details below.

### 6.4.1 Creating Computation Sparsity

A key challenge in using Dropback algorithm [92] to enable energy-efficient training is the fact that it never entirely removes pruned weights: instead, pruned weights have their values returned to their initialization-time values. These are generally non-zero, so no MAC operations are saved, and, because MAC computation accounts for much of the energy during training (cf. Figures 6.1 and 6.15), energy savings are also limited.

To determine how to recover computation sparsity, we first considered the function of the weights during training. We hypothesized that the initial weight values are important during the early iterations when weights have not moved far from their initial state and the accumulated gradients are small compared to the initial weights. Later on, we reasoned, the accumulated gradients are much larger than the initial weight values, and the initial scaffolding could safely be removed.

We therefore examined whether the initial weight values could be gradually decayed to zero so that eventually only the accumulated gradients remain and all pruned weights become zero. We decayed the initial weight values 10% every iteration (decay parameter $\lambda = 0.9$), eventually zeroing them. Figure 6.6 shows how validation accuracy evolves over the course of training compared to a baseline where weights do not decay: neither accuracy nor convergence time are affected.

In this experiment, the initial weight decay scheme results in 80% weights set to zero by iteration 1000 (out of 234,400 total iterations). This means that 60% of computation in 99.5% of iterations can be entirely skipped, potentially resulting in significant energy savings.

### 6.4.2 Choosing Which Weights to Keep

The second key challenge of the original Dropback algorithm is the need to sort all accumulated gradients to determine which weights should be kept and which should be reset to their initial values. A comparison-based sort requires a minimum of $\log_2(n!)$ comparisons in the worst case — 336M comparisons for the relatively compact VGG-S with 15M weights, compared to the 4.3G MACs required for one training iteration with batch 16. Even if the DNN accelerator were modified to support sorting (i.e., to return both indices and values), sorting would take in excess of 1.3M cycles on a 256-PE device.

To overcome this challenge, we considered replacing the target sparsity factor (such as 10×) with a global value threshold $\vartheta$. In this scheme, every computed gradient is tested whether it should be added to the tracked set $T$, and added to $T$ only if it exceeds $\vartheta$. This would reduce the number of comparisons to one per produced gradient (15M for VGG-S).

The question is how to determine $\vartheta$ for each iteration. Dynamic sparse reparametrization [179] accomplishes this via a set-point feedback scheme that adjusts $\vartheta$ every 1,000–8,000 iterations, but this introduces an additional hyperparameter, the initial value of $\vartheta$. Instead, we determine $\vartheta$ dynamically via a streaming quantile estimation technique [265]. To allow for peak update rate (up to 4 per cycle in the last VGG-S CONV layer), we extended the technique to process four updates at once.

The tracking process proceeds as follows:

- If the gradient dimension $\delta_w$ is *not* in the tracked set $T$, $|\delta_w|$ is compared against $\vartheta$. If it is higher, $\delta_w$ evicts and replaces the lowest entry in $T$; otherwise, it is discarded. In either case, $|\delta_w|$ is used to update the quantile estimate.

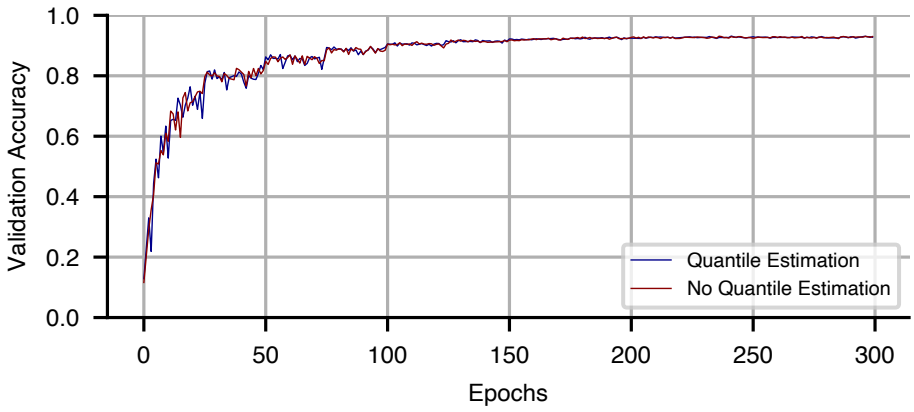



Figure 6.7: Validation accuracy over training epochs when sparse training is used and quantile estimation is used to determine the value threshold $\vartheta$ under which accumulated gradients are discarded, compared to a baseline with initial weight decay and exact sorting (VGG-S on CIFAR-10).

- If $\delta_w$ is tracked, it is added to the stored accumulated gradient $\delta_w^{\text{acc}}$. The quantile estimate is updated with $|\delta_w^{\text{acc}} + \delta_w|$.

In our experiments, we found that the tracking accuracy sensitivity to the values of $\widehat{Q}_q(0)$ and $\varrho$ is negligible, so we use the same values for all experiments rather than treating them as hyperparameters.

To determine the accuracy of this estimate, we trained VGG-S using a sparsity target of 7.5× and streamed the computed accumulated gradients to the estimator. Figure 6.7 shows that while the quantile estimation exhibits minor deviations from ground truth (because different layers have different amounts of sparsity), these estimation errors have no detrimental effect on the validation accuracy of the trained network. Overall, the quantile estimation error results in extra weights being tracked, and reduces the sparsity factor slightly from 7.5× to 5.2×; however, this overhead is much lower than that required to sort all weights or to train a dense network.

Note that selecting weights through quantile estimation is not specific to the Dropback algorithm: separating some fraction of the highest-value or highest-gradient weights is needed by all sparse training algorithms [84, 92, 176, 179].

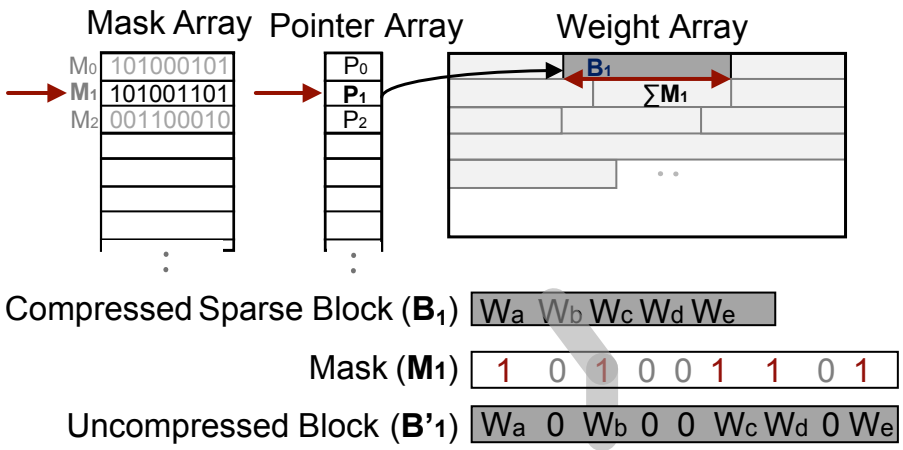



Figure 6.8: The compressed sparse block (CSB) weight representation in Procrustes.

## 6.5 Dataflow and Sparse Data Format

### 6.5.1 Storage and Sparsity During Training

Weights (or, more precisely, accumulated gradients) are always stored compressed using the format described in subsection 6.5.2. Typically, all weight gradients are produced, but most gradients that are not already tracked will not survive the comparison with existing accumulated gradients.

Activations are stored uncompressed for immediate reuse and in a compressed format for long-term reuse. The forward pass reads sparse weight tensor, and produces a dense output activation tensor, which is then immediately reused as inputs to the next layer; the activations are then compressed using a sparse, zero-free format, and reused in the weight update stage. This technique is similar in spirit to Gist [133].

### 6.5.2 Compressed Sparse Weight Representation

To avoid the challenges discussed in subsection 6.3.3, a sparse weight storage format designed for training must support:

(i) iterating through 2D convolution filters across different dimensions in different stages (for CONV layers), and across both rows and columns of weight matrices (for FC layers),

(ii) rotating kernels (for CONV layers) or transposing weight matrices (for FC layers), and

(iii) different kernel sizes in CONV layers.

Procrustes uses a modified compressed sparse block (CSB) format [45] shown in Figure 6.8 to store weights in the on-chip global buffer and external DRAM. Blocks store non-zero values and are variable in size because of sparsity, but correspond to fixed-size regions in the corresponding dense weight space — kernels for CONV layers, square fragments of the weight matrix in FC layers, etc. The region size can vary on layer granularity to support different kernel sizes.

The Procrustes CSB format comprises three components, illustrated in Figure 6.8:

(a) the weight array, which stores variable-size packed weight blocks corresponding to kernels, etc.;

(b) the pointer array, indexed by tensor coordinates, which identifies the weight array location that stores the relevant weight values; and

(c) the mask array, also indexed by tensor coordinates, which stores a mask identifying non-zero value locations in the unpacked block (and therefore also the packed size).

The pointer and mask arrays are decoupled to support different mask lengths for each layer (e.g., different kernel sizes in CONV layers, flexible block sizes in FC layers and during weight update, and so on); in all of our simulations, mask arrays fit in the on-chip GLB.

Because the pointer array is indexed by coordinates in the original (dense) operation space and is decoupled from the compressed contents, the format makes computing kernel addresses straightforward while adapting cleanly to different kernel dimensions. The indirection also makes it easy to determine the density of working sets assigned to each PE: it suffices to subtract pointers of adjacent work tiles. In addition, because blocks are sized to and retrieved on filter granularity, they can be rotated (to be used in the backprop pass) while being fetched from the global buffer to the per-PE register files; similarly, transposition of the weight matrix for the FC layers can be done by transposing subtensors piecewise.

Activations are stored uncompressed for short term reuse (as activations in the next layer) and compressed in CSB format for long-term reuse (forward pass to weight update).

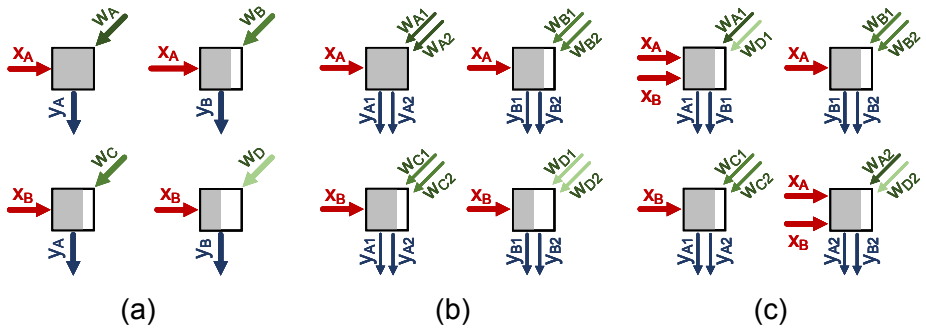
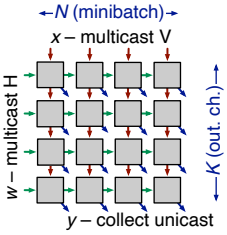



Figure 6.9: Load balancing in the weight-stationary $C, K$ dataflow in a four-PE array: (a) PE workload imbalance (shaded PEs) due to different weight sparsities (shaded arrows); (b) PE workloads and the corresponding weight ($w$) and partial sum ($y$) tiles split in half across the $K$ dimension (note the thinner arrows); (c) half-tiles exchanged between the top-left and bottom-right PEs for load balancing. Activations must be sent on both rows and columns, and require twice the buffer space in the PEs.

### 6.5.3 Load Balancing and Dataflow

First, every work tile (a) is cut into two halves along one of the tile dimensions (b); because sparsity is almost certainly uneven within the tile, the two halves will likely have different densities. Next, the halves are sorted according to density, and half-tiles are matched starting from opposite ends (c): the sparsest half-tile is matched with the densest half-tile, and so on. This ensures that each newly formed tile is as close as possible to the average density across all PE work tiles (d).

However, naively applying this rebalancing scheme to the entire PE array without changing the dataflow would impact on-chip communications patterns and require a complex interconnect. Figure 6.9 demonstrates this on the weight-stationary $C, K$ dataflow on a 4-PE array. In pane (a), input activations are broadcast horizontally ($x_A$ in the top row and $x_B$ in the bottom row), partial sums are accumulated vertically ($y_A$ in the left column and $y_B$ in the right column), while the weights are unicast (as in Figure 6.3); however, because the weights have different levels of sparsity (shaded arrows), the PEs have different amount of computation (shaded PEs). In pane (b), each PE's workload is cut in half as discussed above; each weights tile ($w_A$ and $w_B$) is also split in half (e.g., into $w_{A1} + w_{A2}$ and $w_{B1} + w_{B2}$, note the thinner arrows), as are the corresponding partial sums ($y_A$ and $y_B$). Finally,




| | fw | bw | wu |
|---|---|---|---|
| (H)orizontal | $w$ | $w$ | $\partial\mathcal{L}/\partial w$ |
| (V)ertical | $x$ | $\partial\mathcal{L}/\partial x$ | $x$ |
| (U)nicast | $y$ | $\partial\mathcal{L}/\partial y$ | $\partial\mathcal{L}/\partial y$ |

$w$ weights $x$ input activations
$y$ output activation partial sums

Figure 6.10: Mappings and dataflows that spatially distribute the minibatch across one dimension of the PE array.

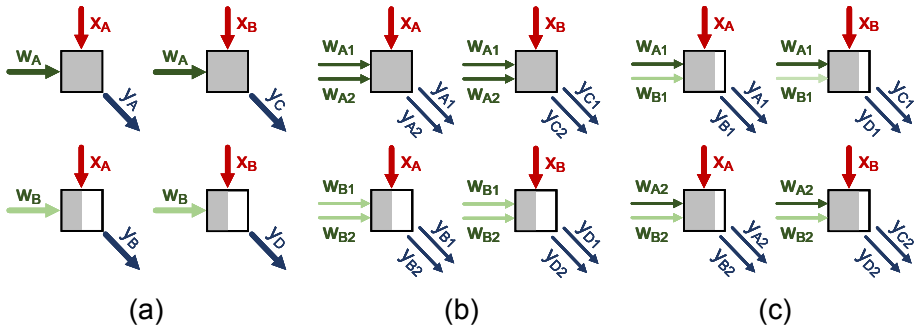



Figure 6.11: Load balancing in the proposed $K, N$ dataflow in a four-PE array: (a) PE workload imbalance (shaded PEs) due to different weight sparsities (shaded arrows); (b) PE workloads and the corresponding weight ($w$) and partial sum ($y$) tiles split in half across the $K$ dimension (note the thinner arrows); (c) half-tiles exchanged between the top-left and bottom-right PEs to load-balance across $K$. Each input activation tile is still sent to only one column.

in pane (c), the workload halves are balanced across the PE array, so that the top-left and bottom-right PEs swap half their workloads; this, however, means that all input activations ($x_A$ and $x_B$) must now be sent to both columns and rows, requiring more bandwidth and a more complex interconnect, and double the activations must be buffered at the target PEs. The $P, Q$ input-stationary dataflow faces similar challenges in the weight update pass (cf. Figure 6.2) and requires unicasting two of the three datatypes.

Procrustes addresses both of these problems by leveraging a simple observation: training is typically done across a minibatch of 32–64 samples rather than on single items [36, 171].[4]

Because a training accelerator does not need to support single-sample inference, the minibatch dimension can be used to distribute work tiles across one dimension of the PE array. The other dimension can then be safely chosen to be a dimension where sparsity exists — e.g., the input or output channel dimensions ($C$ or $K$) with weight sparsity. Because only one dimension is sparse, and that dimension corresponds to spatial reuse, the load balancing process needs to be applied only to one dimension of the PE array (i.e., the dimension opposite to the spatial reuse pattern, here $N$).

Figure 6.10 illustrates how a $K, N$ mapping (output channel, minibatch) with load balancing across the output channel ($K$) dimension preserves the single-dimension dataflow properties (cf. Figure 6.3) during the forward pass. Weights are now the same across the minibatch and are multicast across the horizontal dimension of the PE array, partial sums are collected across the vertical dimension, and input activations vary across both dimensions and so are unicast.

A detailed example is shown in Figure 6.11. As in Figure 6.9, pane (a) shows the unbalanced workload, pane (b) shows each PE's workload (and consequently the weight and partial sum tiles) cut into half, and pane (c) shows the PE array after load-balancing along the $K$ dimension. Observe that, in contrast to Figure 6.9, the load-balanced dataflow in pane (c) has the same on-chip interconnect communication patterns and requires the same interconnect bandwidth as the unbalanced dataflow in pane (c).

## 6.6 Architecture

The overall hardware architecture of Procrustes is based on 2D PE array where each PE has a local register file (RF) and all PEs share an on-chip

[4]Minibatches in the 1,000s allow faster training on large multi-GPU clusters but can incur some accuracy cost [19, 98].

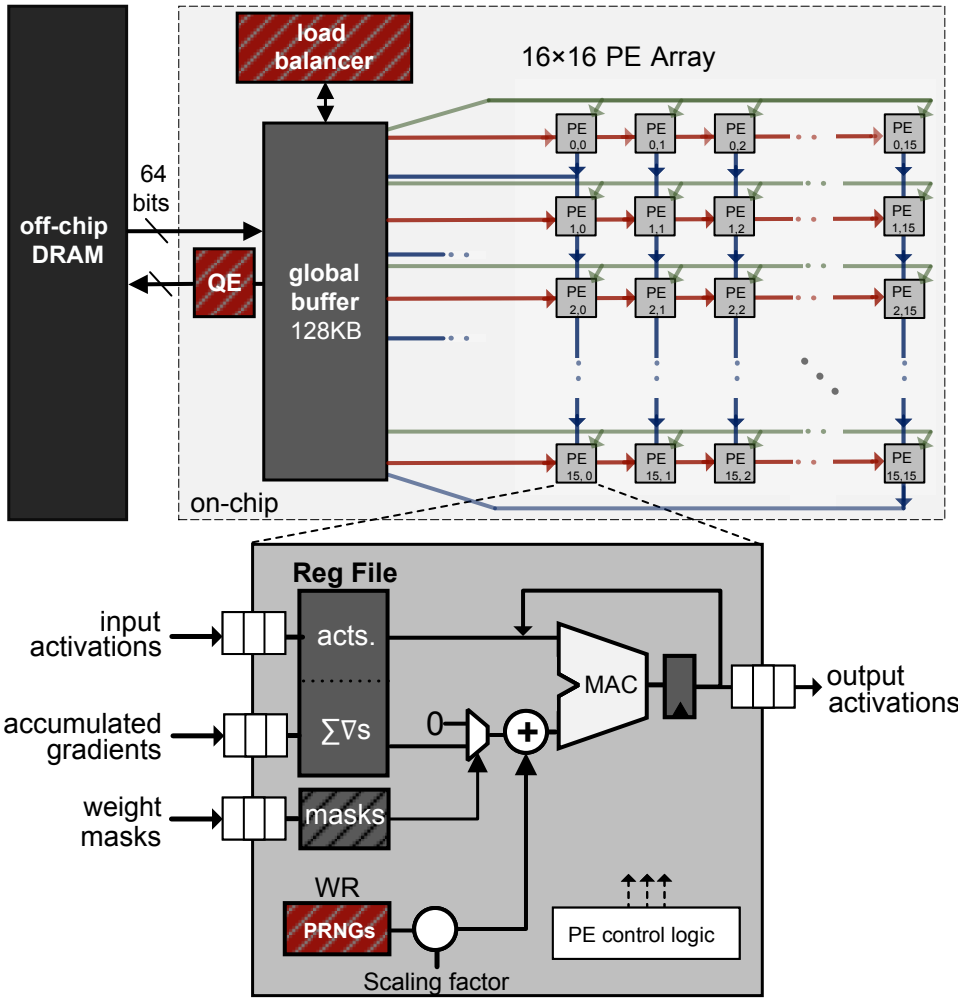



Figure 6.12: Procrustes system architecture. The WR module is added to recreate initial weights for Dropback-style training, the QE unit is added to support quantile estimation, and the load balancer is added to support work tile re-balancing.

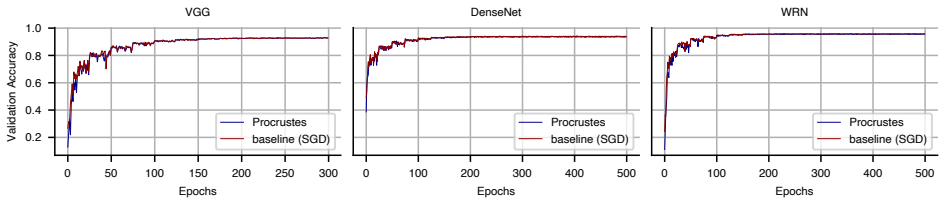



Figure 6.13: Validation accuracy over training time for Procrustes and the unpruned baseline (SGD) on CIFAR-10: (left) VGG-S, (centre) DenseNet, and (right) WRN-10-28.

global buffer (GLB); an off-chip DRAM completes the memory hierarchy. PEs are interconnected via three simple interconnects: two support one-dimensional traffic flows in the horizontal and vertical directions, and one supports unicast traffic to any PE in the array. Because Procrustes focuses on training, we use 32-bit floating point MAC units in the PE datapath, but the design can be used with any datatype.

The design is illustrated in Figure 6.12, with differences from the baseline accelerator dashed. Procrustes places one global quantile estimation unit (QE) between global buffer and the external DRAM; the QE unit monitors accumulated gradients flowing from the GLB to DRAM and discards all except those above the target sparsity quantile.

In addition, each PE contains a weight recomputation unit (WR) responsible for generating the initial weight values. The WR accepts a weight index and generates a 32-bit integer initial value for the relevant weight. It consists of 3 xorshift [170] pseudo-random generators (RNGs) whose outputs are added to produce an approximately Gaussian output. Note that, unlike conventional RNG, the WR unit does not contain hidden state, and is purely a function of its seed and the weight index. The "RNG" output is then scaled using an integer multiplier; this this enables popular initialization formulæ like Xavier [91] or Kaiming [111], and allows the initial weights to be decayed. Finally, the scaled value is converted to FP32 and added to the accumulated gradient retrieved from weight storage if the weight is tracked, or to zero if the weight has been pruned.

## 6.7 Methodology

We re-implemented the baseline Dropback training algorithm [92] using PyTorch [196] and verified the reported training sparsity levels and accuracy re-

| Baseline dense accelerator | |
|---|---|
| PEs | 256 (16×16) |
| datatype | 32-bit floating-point |
| pruning type | none |
| interconnect | 3× 1D-flow interconnect |
| global buffer | 128 KB |
| local buffer (RF) | 1 KB per PE |
| dataflow | optimal (via Timeloop+Accelergy) |
| **Procrustes modifications** | |
| pruning type | lowest accumulated gradients |
| pseudo-RNG | xorshift [170], one per PE |
| quantile estimator | DUMIQUE [265], max 4 requests / cycle |
| dataflow | optimal spatial-minibatch dimension |

Table 6.1: Hardware configurations for the baseline dense training accelerator and Procrustes sparse training accelerator.

| model | dataset | dense size | sparse size | sparsity | #ep | dense accuracy | pruned accuracy |
|---|---|---|---|---|---|---|---|
| Densenet | CIFAR-10 | 2.7M | 692k | 3.9× | 340 | 94.2% | 93.7% |
| WRN-28-10 | CIFAR-10 | 36M | 8.3M | 4.3× | 462 | 96.0% | 96.1% |
| VGG-S | CIFAR-10 | 15M | 2.9M | 5.2× | 236 | 93.0% | 93.1% |
| MobileNet v2 | ImageNet | 3.5M | 0.35M | 10× | 131 | 70.98% | 71.13% |
| ResNet18 | ImageNet | 11.7M | 1M | 11.7× | 81 | 69.17% | 69.31% |

Table 6.2: Sparsity achieved using the Procrustes training scheme for the CNNs tested, together with weight footprint and the final accuracy compared to the dense baseline.

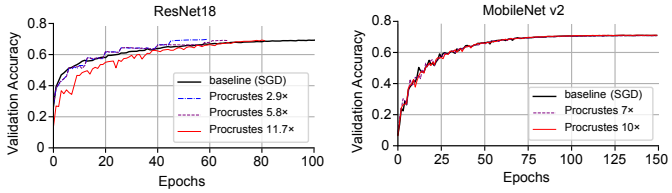



Figure 6.14: Validation accuracy over training time for Procrustes and the unpruned baseline for ResNet18 (left) and MobileNet v2 (right) on ImageNet.

| model | dataset | dense size | dense MACs | sparse size | sparse MACs | sparsity |
|---|---|---|---|---|---|---|
| Densenet | CIFAR-10 | 2.7M | 528M | 692k | 157M | 3.9× |
| WRN-28-10 | CIFAR-10 | 36M | 4G | 8.3M | 863M | 4.3× |
| VGG-S | CIFAR-10 | 15M | 269M | 2.9M | 113M | 5.2× |
| MobileNet v2 | ImageNet | 3.5M | 301M | 0.35M | 75M | 10× |
| ResNet18 | ImageNet | 11.7M | 1.8G | 1M | 359M | 11.7× |

Table 6.3: Sparsity achieved using the Procrustes training scheme for the CNNs tested, together with weight footprint and MAC reduction compared to the dense baseline.

sults; we then implemented the initial weight decay and quantile-estimation extensions needed for Procrustes.

We evaluated Procrustes on five CNNs: ResNet18 [112] (11.7M weights) and MobileNet v2 (3.5M weights) applied to the ImageNet image classification task [209], as well VGG-S [272] (a 9.2× reduced version of VGG-16 with 15M weights), WRN-28-10 [273] (36.5M weights), and a small Densenet [123] (growth rate 24, 3 blocks × 10 layers, for a total of 2.7M weights), all CIFAR-10 [148].

To determine optimal mappings and dataflows, we extended Timeloop tool [192] to support sparse weight masks (retrieved from our PyTorch model), model sparse computation, account for sparse encoding overheads, and accurately reflect latency due to load imbalances. We also used Timeloop to determine cycle-level latency; to determine energy costs, we use energy access cost provided in Accelergy [256] with its default 40nm library. We modelled all layers of all networks and all stages of training (forward, backward, and weight update).

As a dense baseline, we used a 2D PE array architecture with 16×16 PEs, adapted to the 32-bit floating-point precision commonly used in training; we used Timeloop to determine the optimal tiling and dataflow. Hardware modules not present in the baseline were implemented in Verilog RTL and synthesized using Synopsys DC in the 45nm FreePDK process. Accelerator configuration details are shown in Table 6.1.

### 6.7.1 Pruning Ratios and Accuracy

Table 6.2 shows the sparsity factors achieved while maintaining the same accuracy as the corresponding dense (unpruned) network using the Procrustes

sparse training algorithm. Depending on the network, our training scheme achieves 3.9×–11.7× weight sparsity without compromising accuracy.

Importantly, achieving unpruned-level accuracy does not require additional convergence time. Figure 6.13 demonstrates this on the VGG-S, DenseNet, and WRN, all on CIFAR-10. Figure 6.14 demonstrates the same effect on ResNet18 trained on ImageNet at various weight pruning ratios. Overall, Procrustes converges and reaches state-of-the-art accuracy as quickly (or faster) than the baseline unpruned network.

### 6.7.2 Energy Savings and Speedup

Figure 6.15 shows the energy savings obtained by training with Procrustes across several CNNs. Most of the energy is saved by performing fewer MAC operations; because training is most often done on FP32 values, MACs dominate the energy usage. Intra-PE register file (RF), global buffer (GLB), and DRAM access energies are also substantially reduced, but account for less of the baseline energy expenditure, and therefore contribute less to the overall savings.

The figure also illustrates that Procrustes can transform higher sparsity ratios into bigger energy savings: ResNet18, which has the highest pruning factor (11×), saves the most energy compared to the dense baseline (3.26×), while WRN has the best speedup (4×). MobileNet v2 benefits less in energy because its depth-separable convolutions limit reuse and so comparatively more energy is spent on DRAM accesses; however, Procrustes still trains it with 2.39× less energy than the dense baseline, and almost as much speedup as WRN (3.88× faster than dense).

For most networks, the forward and back-propagation passes offer more energy savings; this is because those passes can take advantage of weight sparsity, which is generally higher than activation sparsity. VGG-S demonstrates a less common case where the weight sparsity is concentrated in the layers that perform relatively few MACs, so the activation sparsity leveraged by the weight-update phase actually saves more operations.

Overall, Procrustes is effective in converting training-time sparsity to energy savings.

### 6.7.3 Mapping and Dataflow Choice

Figure 6.16 shows how energy expenditure varies with different spatial partitioning schemes. Sparsity enables energy improvement across all phases and all mappings. Because the number of MAC operations and the mem-

ory hierarchy are the same across the different mappings, the lion's share of the energy use is the same across the different dataflows, and variations are negligible. This is in agreement with prior work that also reported negligible impact of the chosen dataflow on the energy during inference [262].

This finding enables us to select spatial partitioning that results in the best performance (i.e., shortest execution time).

Figure 6.17 shows how execution times vary when the working set is mapped to the PE array using different spatial partitioning schemes; all schemes can be implemented using the simple network topology shown in Figure 6.12 except the weight-stationary $C, K$ scheme, which requires a complex network to load-balance PE working sets across the entire chip. The partitioning schemes that distribute the minibatch dimension along one of the PE array dimensions ($C, N$ and $K, N$) are the fastest mappings because they are able to achieve effective load balancing and good utilization across all layers of the CNNs; $K, N$ performs slightly better because it offers slightly higher utilization in the first network layer. The $C, K$ scheme performs less well even though it requires a more complex interconnect, largely because it is inefficient on layers that have few channels. The activation-stationary $P, Q$ scheme does not require load-balancing in the forward and back-propagation phases, is hard to load-balance during the weight update phase, and has low utilization when activation tensors are relatively small; it is overall the slowest mapping.

Procrustes uses the overall fastest $K, N$ scheme for all phases of training.

## 6.8 Evaluation Results

### 6.8.1 Scalability

Figure 6.18 shows how Procrustes scales when the PE array size is quadrupled from 256 cores (16×16) to 1024 cores (32×32); the global buffer size is doubled over the 256-core size (a factor of $\sqrt{2}$). Overall, energy is very similar same for all dataflows / passes because the number of MAC operations is the same. Latency scales near ideally (3.9× on 4× the cores) in the $K, N$ mapping used by Procrustes. Other mappings (especially activation-stationary $P, Q$) do not scale as well since they trade off PE array utilization to retain spatial reuse.

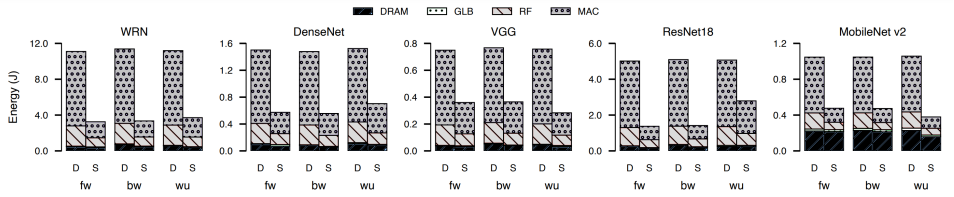



Figure 6.15: Energy breakdown of using KN dataflow for (left) WRN-10-28, (middle left) DenseNet, (middle) VGG-S, (middle right) ResNet18, and (right) MobileNet v2. Lower is better. K = output channel dimension; N = minibatch dimension. S = sparse; D = dense. fw = forward pass; bw = backward pass; wu = weight update phase.

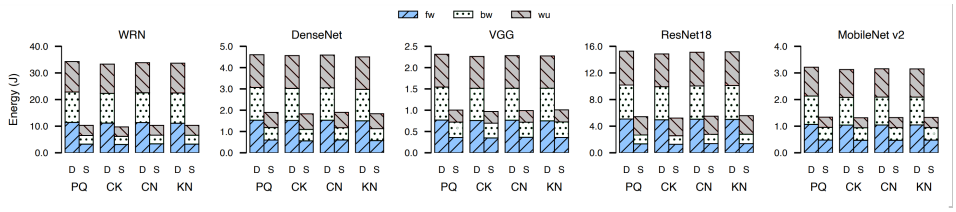



Figure 6.16: Energy Comparison across different dataflows for (left) WRN-10-28, (middle left) DenseNet, (middle) VGG-S, (middle right) ResNet18, and (right) MobileNet v2. Lower is better. C = input channel dimension; K = output channel dimension; P and Q = output activation dimensions; N = minibatch dimension. S = sparse; D = dense. fw = forward pass; bw = backward pass; wu = weight update phase.

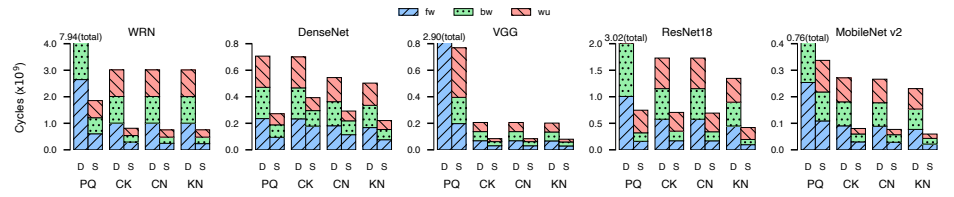



Figure 6.17: Training latency across different dataflows for (left) WRN-10-28, (middle left) DenseNet, (middle) VGG-S, (middle right) ResNet18, and (right) MobileNet v2. Lower is better. C = input channel dimension; K = output channel dimension; P and Q = output activation dimensions; N = minibatch dimension. S = sparse; D = dense. fw = forward pass; bw = backward pass; wu = weight update phase.

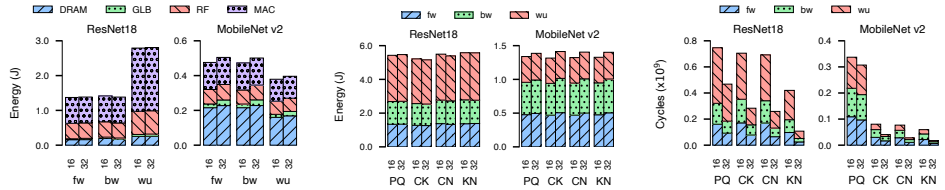



Figure 6.18: Scalability of Procrustes on 16×16 (256) to 32×32 (1024) cores on ResNet-18 and MobileNet v2 classifying ImageNet configured as in Figs. 6.15–6.17. Energy differences are is negligible as the workload is the same. Speedup scales best for the Procrustes mappings (CN and KN) because other mappings trade off utilization for reuse.

| **Component** | **Power** $(mW)$ | **Area** $(\mu m^2)$ |
|---|---|---|
| **Per-PE area: Procrustes overheads italicized** | | |
| FP32 MAC | 7.29 | 18,875.72 |
| Register File | 15.61 | 198,004.71 |
| *PRNG* | *0.35* | *1,920.84* |
| *Mask Memory* | *2.65* | *44,932.66* |
| **System area: Procrustes overheads italicized** | | |
| Global Buffer | 73.74 | 17,109,596.5 |
| *Quantile Engine* | *1.38* | *9,861.4* |
| *Load Balancer* | *2.05* | *8,725.23* |

Table 6.4: Silicon area costs and overheads (synthesis using Synopsys DC with the FreePDK 45nm library). For fairness, the power estimates assume the same dense computation (i.e., no sparsity).

### 6.8.2 Silicon Area Overheads

The silicon area and power overheads of Procrustes are detailed in Table 6.4. Despite the RNG initial weight recomputation module being included in every PE, its area and power pale in comparison to the FP32 MAC unit which all PEs include.

Overall, the Procrustes accelerator has an area overhead of 14% over an equivalent dense accelerator, and consumes 11% more power when executing the same *dense* workloads. Both are a small price to pay for the 2.27×–3.26× energy savings offered by sparse training.

### 6.8.3 Generality

Procrustes is the first sparse training accelerator to combine substantial sparsity ratios, 2.27×–3.26× energy savings, and up to 4× speedups while maintaining state-of-the-art accuracy of the trained networks. While in this work we use Procrustes to extend the Dropback training algorithm, the quantile estimation and spatial-minibatch dataflow insights apply to all existing — and likely many future — sparse training algorithms.

## 6.9 Summary

This chapter introduces Procrustes, a sparse DNN training accelerator that produces pruned models with the same accuracy as dense models without first training, then pruning, and finally retraining, a dense model. The sparsification process considers the entire set of gradient values to decide on what weights are important to keep, and what can be dropped.

Procrustes relies on three key techniques. First, it adapts an existing training algorithm to create computation sparsity that can be converted into energy savings. Next, it replaces the sorting step present in nearly all sparse training algorithms with hardware-friendly, computationally simple quantile estimation. Finally, it leverages a novel load-balancing scheme that converts sparsity into speedup, and proposes a novel dataflow that enables load balancing without significant changes to the on-chip interconnect.

# Chapter 7

# Related Work

## 7.1 Cache Compression

In chapter 3 and chapter 4, we present two novel cache compression mechanisms that we develop to overcome limitations of prior methods. Prior work on cache compression can generally be categorized into three categories based on their compression granularities: (i) inter-block data compression, (ii) intra-block data compression, and (iii) techniques that do not operate at block granularity. Below, we outline past proposals in all of these categories, and discuss prior work on orthogonal ideas on effective replacement policies with compression.

### 7.1.1 Inter-Block Data Compression

Inter-block data deduplication techniques leverage the observation that many cache blocks are either entirely zero [76, 78, 200] or are copies of other blocks that concurrently reside in the cache [59, 70, 117, 231, 242]. Instead of storing several identical copies, they aim to store only one copy of the block in the cache, and propose techniques to point the redundant data entries to this single copy.

To address the inter-block redundancy at the cache level, Dedup [242] modified a conventional cache to store one copy of the redundant data and allow multiple tags to pointing to the unique copy. The key challenge here is that probing the entire cache to search for duplicates is impractical (unlike compressing a single cacheline, as B$\Delta$I does).

To overcome this limitation, Dedup uses a "hash table" that stores (say) 16-bit fingerprints of 64-byte memory blocks and their locations in the data array. While a "hit" must be verified against the actual 64-byte block to avoid false matches, in practice collisions are rare. Because cached values have some temporal locality, using a limited-size hash table with the most recently used fingerprints (say up to 1024 hashes) covers most of the duplication in typical working sets [242].

In addition to the limitations due to the exact-match requirement, Dedup

has two performance challenges. One is that data insertion involves a reference to the hash-table followed by a reference into the LLC data array to verify the exact contents of the memory block. Another limitation is that evicting a deduplicated memory block from the data array requires evicting *all* tags that point to it; in turn, this means that the tag array entries must contain two pointers to form a doubly-linked list for each deduplicated memory block value.

### 7.1.2 Intra-Block Data Compression

For some applications, data values stored within a block have a low dynamic range resulting in redundancies [21, 200, 250]. Prior work categorized these into (a) repeated values (especially zeros) repeated in a data block, and (b) *near* values with the same upper data bits and different lower bits.

One way to reduce redundancy within the memory block is to capture the replicated data in dictionary entries and then point to that entry when new replicated data is presented. Frequent pattern compression [21] does this on a word-by-word basis by storing the last 16 observed values as a dictionary. Similarly, [250] uses an LZ77-like compression algorithm by reading through the input data word by word and constructing a dictionary of observed sequences. The authors of [129] propose a small cache placed alongside the L1data cache to store memory locations with narrow values; this compactly stores each 32-bit word at 1-, 2-, 4-, and 8-bit granularity.

Another method to reduce redundancy of nearly identical values is to try to separate repeated parts of values from distinct lower bits in a memory block. DISH [191] extracts distinct 4-byte chunks of a memory block and uses encoding schemes to compress them, with dictionaries potentially shared among a few contiguous blocks. It uses a fixed-width pointer that points to one of the $n$ dictionary entries: i.e., a cache block is encoded as a dictionary, some fixed-width pointers, and some lower-bit deltas for each 4-byte chunk.

B$\Delta$I [200] uses one word-granularity "base" value for each compressed cache block, and replaces the other words in the block with their distances from the base value. B$\Delta$I can compress zero lines, as well as various combinations of base value and offset sizes; the type of compression selected is encoded in the tag entry metadata. A data block is logically divided into eight fixed-size segments, and compressed blocks are stored as multiple segments allocated at segment granularity.

$SC^2$ [26] uses Huffman coding to compress memory blocks, and recomputes the dictionary infrequently, leveraging the observation that frequent values change rarely. HyComp [25] combines multiple compression algo-

rithms and dynamically selects the best-performing scheme, based on heuristics that predict data types. Bit-Plane Compression [143] targets homogeneous arrays in GPGPUs to both improve the inherent data compressibility and to reduce the complexity of compression hardware over B$\Delta$I by compressing the deltas better. To reduce tag overhead of the compressed cache, DCC [221] and SCC [218] use "superblocks" formed by grouping adjacent memory blocks in the physical address space. More recently, YACC [219] was proposed to reduce the complexity of SCC by exploiting spatial locality for compression.

Broadly, intra-block methods are useful in compressing one block or possibly a "superblock" of contiguous memory blocks. However, unlike Thesaurus, they do not consider value redundancy among different non-contiguous memory blocks at far-away addresses, which still leads to repeated (albeit potentially compressed) data values in different parts of the cache.

### 7.1.3 Non-Block-Granularity Compression

Unlike scientific applications, whose working sets are often dominated by arrays of primitive-type values, many general-purpose applications traverse and operate on blocks. Based on this insight, Cross-Block-Compression algorithm (COCO) [243] uses data structure blocks (rather than cache blocks) as the unit of compression. The authors also present the first compressed memory hierarchy designed for block-based applications.

Our cache compression proposal, Thesaurus (chapter 4) is able to capture the redundancy across objects stored in the cache memory. Thesaurus does not require cachelines to be filled with a particular datatype, as it does not impose any limitation on the structure of the data being compressed. Nevertheless, if objects with various fields are stored across multiple lines in the cache, Thesaurus can form clusters of similar cachelines (storing all or parts of these objects) and efficiently compress them.

### 7.1.4 Replacement Policies With Compression

Prior works have also looked at the impact of compression on cache replacement policy. ECM [31] reduces the cache misses using Size-Aware Insertion and Size-Aware Replacement. CAMP [198] exploits the compressed cache block size as a reuse indicator. Base-Victim [86] was also proposed to avoid performance degradation due to compression on the replacement. These proposals are effective for intra-cacheline compression, but do not consider the inter-cacheline interactions present in Thesaurus and Dedup [242].

## 7.2 DNN Compression

In chapter 5 and chapter 6, we present compression methods to efficiently reduce activation and weight memory footprints. In what follows, we describe the contributions of prior work towards reducing storage and energy costs through lossy and lossless compression, both of weights and of activations of DNNs

Techniques for weight compression usually explore redundancy in the weights and try to approximate or remove the redundant and uncritical parameters of deep CNNs [55, 67, 72, 84, 96, 101, 103, 106, 107, 110, 113, 161, 172, 179, 183, 184, 195, 204, 205, 235, 245, 254, 255, 258, 271, 278, 280, 284**?** ]. Techniques on activation compression [22, 32, 62, 64, 87, 87, 118, 124, 133, 140, 141, 165, 206, 207, 225, 249, 266] try to save the memory footprint of the intermediate values as these activation maps can occupy up to 90% of the GPU-side memory allocations on some networks [205, 232].

### 7.2.1 Weight Compression

Most of the work to date [67, 92, 101, 110, 161, 183, 184, 195, 284] has focused on shrinking the size of the model parameters as a way to increase memory efficiency. Weight pruning cuts the less-important connections in networks [84, 92, 107, 113, 280] and results in sparse models, shrinking the model size by an order of magnitude.

Most proposed sparse training algorithms gradually increase sparsity during the training process. The lottery ticket algorithm [84] prunes 20% of the network every 50,000 training iterations by removing the lowest-magnitude weights; the authors report 5–10× model size reduction on CIFAR10 targets. Eager Pruning [278] follows a similar magnitude-based approach, but adds a feedback loop and a checkpoint-based rollback scheme to avoid overpruning; maintaining top-1 accuracy on ImageNet, it can prune ResNet50 2.4× (25.6M→10.8M weights) by removing 0.8% of the weights every 24,000 iterations. Both approaches rely on sorting all weight values to determine which weights to keep. Dynamic sparse reparametrization [179] follows a similar magnitude-based pruning approach, pruning ∼200,000 parameters from ResNet50 every 1,000–8,000 iterations, but allows a fraction of new weights to regrow after each pruning step. It avoids the need to sort all weights by using a value threshold adjusted via a set-point feedback loop whenever the network is pruned; however, the initial value of this threshold becomes a hyperparameter. This method prunes ResNet50 3.5× (25.6M→7.3M) with some top-1 accuracy loss on ImageNet.

Dropback [92] prunes the network from the beginning: only a fixed percentage of the parameters (e.g., 10%) are ever allowed to change. In every iteration, only the weights with the highest accumulated gradient survive (which again requires sorting), on the theory that this represents learning better than magnitude during early iterations; the pruned weights are reset to their initial values rather than to 0. With Dropback, ResNet18 can be pruned 11.7× (11.7M→1M) while maintaining top-1 accuracy on ImageNet. Procrustes adapts Dropback to the needs of an efficient hardware implementation, removing the requirement for sorting and decaying initial weights to 0 to create extra computation sparsity.

[235] explored the redundancy among neurons, and proposed a data-free pruning method to remove redundant neurons.

Network quantization compresses the original network by reducing the number of bits required to represent each weight. [245] showed that 8-bit quantization of the parameters can result in significant speed-up with minimal loss of accuracy. The work in [103] used 16-bit fixed-point representation in stochastic rounding based CNN training, which significantly reduced memory usage and float point operations with little loss in classification accuracy. In [258], it was shown that Hessian weight could be used to measure the importance of network parameters, and proposed to minimize Hessian weighted quantization errors in average to cluster parameters.

Weight clipping methods [161, 184] help quantization by limiting the data range. Weight sharing and quantization methods assume that many weights have similar values, and can thus be grouped in order to reduce the number of free parameters. The work in [55] proposed a hashing technique to randomly group the connection weights into a single bucket and then fine-tune the model to recover from the accuracy loss. The most important property of the hashing trick is, arguably, its approximate preservation of inner product operations. Other work also proposed grouping the weights using K-Means [255]. [96, 254] applied K-Means scalar quantization to the parameter values. Lower bit-width weights like binarization [204] has also been used for the model compression. Extending this, [284] used ternary quantization learned from the given data. Recently, [172] conducted the network compression based on the float value quantization for model storage. Proposed work on quantization sometimes needs significant manual effort for each network to choose the right quantization precision and method. The deep compression method in [106] removed the redundant connections and quantized the weights, and then used Huffman coding to encode the quantized weights.

Reducing weight dimensions by low-rank approximation saves storage

and simultaneously reduces time complexity during training and testing. Most methods [72, 132] approximate a tensor by minimizing the reconstruction error of the original parameters. However, these approaches tend to accumulate errors when multiple layers are compressed sequentially, and the output feature maps deviate far from the original values with the increase of compressed layers [271].

We propose Procrustes in chapter 6 with the energy efficiency in mind. Prior work target a context where pruning occurs after training: a model is first trained with the full parameter set, then pruned, and finally re-trained to recover accuracy [107] whereas in Procrustes, the compression happens during the training process from the very beginning. Also, most [84, 92, 278] require sorting all weights to determine the parameters to retain; with weight counts in the tens of millions, sorting is an expensive proposition. Several [179, 278] achieve only small pruning factors and suffer accuracy loss. Procrustes also does not give up on the computation sparsity as in [92] which is a significant drawback for training.

### 7.2.2 Activation Map Compression

Unlike model compression, surprisingly few work consider intermediate value compression. This can most likely be explained by the fact that intermediate values need to be compressed for every input as opposed to offline model compression. A major drawback of these methods is that they don't directly translate to memory savings on recent networks, as their major bottleneck is the size of activations. The goal here is to decrease computation, memory storage and bandwidth requirements during inference or training. The memory and computations required for DNNs can be excessive for low-power deployments.

In general, similar to model compression, there are some compression work based on pruning, quantization and clustering.

There has been some work on activation map pruning with the same insight as weight pruning [108] (i.e., to remove non-informative wights from the network), which tried to introduce more sparsity to the intermediate activation map values. Recently, [87] proposed a three-stage compression and acceleration pipeline that sparsifies, quantize and entropy encode the activation maps of CNNs. The sparsification step uses L1-norm and increases the number of zero values leading to model acceleration on specialized hardware. The linear quantization and sparse encoding stages lead to compression by effectively utilizing the lower entropy of the sparser activation maps.

Other work in pruning proposed to prune the entire feature maps of a

channel such as [124, 165, 266] with respect to some threshold which yield more compact networks. [165] performed channel-level pruning by attaching a learnable scaling factor to each channel and enforcing $L1-norm$ on those parameters during the training.

Some methods are developed around the idea of quantization. The base floating-point model is converted to the approximate quantized representation, and therefore the intermediate values; then, the quantized model is retrained to restore accuracy.

Proteus [141] suggests that there is a significant variance in the precision needed for both activations and the weights across different networks and layers. So, quantizing the values to only a few bits, which can be highly beneficial in over parameterized networks, can reduce data traffic and storage footprint needed by DNNs significantly. The work in [133] extends this insight and shows that the bit-width needed for every layer, as well as each stage of the computation (i.e., forward and backward propagation) are different and can vary a lot; therefore, they used layer-specific encoding schemes (lossy and lossless) to code backward path activation maps with as few as 8 bits while keeping forward activation in a 32-bit single-precision format. Other work focused on bit-serial operation on the values, such as [22, 140, 225] which significantly reduce the memory bandwidth needs and exploit the variable bit-width need of values in the network. Recently, ShapeShifter [150] adjusted the data type width at a much finer granularity by grouping set of 16 values together.

Wen et al. [249] added a structured sparsity regularizer on each layer to reduce trivial filters, channels or even layers. In the filter-level pruning, all the above works used L1 or L2-norm regularizers.

In many cases, taking a model trained for full precision and directly quantizing it to 8-bit precision, without any re-training, can result in a relatively small loss of accuracy [94, 131, 146]; however, the accuracy degradation can be significant on smaller bit-widths [32]. Many attempts have been made to ameliorate this effect, usually by training the model with quantization constraints [61, 95] or modifying the network structure [258]. The work in [146] introduces a dynamic range quantization scheme for activations using calibration data to perform post-training quantization (PTQ).

Other work has focused on how to estimate the dynamic range of a layer with a given activation distribution, ranging from naive min/max statistics to more advanced methods utilizing statistical analysis. [207] duplicates and halves outliers (large values affecting the dynamic range) in weight or activation values to move them to the center of the distribution, and effectively shrink the range for better quantization resolution. ACIQ [32] approximates

the optimal clipping analytically from the distribution of layer by minimizing MSE, similar to the minimum mean squared error problem formalized in OMSE [62].

[100] compress activation maps by projecting them down to binary vectors and then applying a nonlinear dimensionality reduction technique. However, the method modifies the network structure and it has only been shown to sparsify activations slightly better over simply quantizing activation maps.

Several recent training-free methods cover zero-shot scenarios, where none of the training data is accessible during the PTQ. DFQ [184], focusing on the weights, equalizes the ranges of pre-trained models and corrects biases of the quantization error using batch normalization parameters. ZeroQ [48] creates a calibration dataset by distilling the data distribution from the statistics of batch normalization layers, stored at full precision.These methods suffer from a severe accuracy degradation at lower precision, and their applications are limited to networks where pre-trained batch normalization layers store statistics of training dataset. In contrast, our proposed In-Deployment method Channeleon does not need a full precision model and does not require any distilled data.

The work to compress DMA engines [206] examines three lossless activation map compression techniques: run-length encoding, zero-value compression, and zlib compression. The first two are hardware-friendly, but only achieve competitive compression when sparsity is high, while zlib cannot be used in hardware in practice due to its high computational complexity. Other work in [87] proposes an entropy coding algorithm that leverages on the sparsity and statistical properties of the activation maps to compress them. This algorithm can stand on its own in scenarios where lossy compression is deemed unacceptable, but it is challenging to perform it cheaply in hardware. [133] proposed two lossless encoding to reduce values taken from ReLU layers followed by a specific layers and represent them by as low as 1 bit. Recently, Buddy Compression [64] applied Bit-Plane Compression [143] to the values that has been shown to have high compression ratios for GPU benchmarks when applied for DRAM bandwidth compression. Although this method shows promising results, its focus is on values in GPU and cannot be used directly for accelerator design.

Non-uniform quantization was suggested by [33] and V-Quant [194]. V-Quant suffers a huge accuracy drop in low bit-widths and needs retraining before it can be used for quantized inference. PWLQ [81] breaks the quantization range of weights into non-overlapping regions for each layer while uniformly quantizing activations. Other work used clustering in order to compress the values in DNNs. For example, in [106, 201] parameters are

approximated using K-Means. Because weights are static, this method can be applied off-line, whereas activations are dynamic and the high cost of K-Means prevents it from being computed once for every input image. ACP [52] uses DBScan to identify similar channels and prunes them; a retraining phase follows to recover lost accuracy. [201] proposes clustering of weights and activations to quantize selected layers in AlexNet [147] and VGG16 [230], with better accuracy than linear quantization methods. However, QUENN is applied to older models, and does not quantize all the layers. Moreover, the cost of applying naive K-Means is not studied in this work.

[118] performs channel-wise quantization of activations and layer-wise quantization of weights to compress the image super-resolution networks with and without batch normalization.

We propose Channeleon in chapter 5 which eliminates the need for accessing the training data while maintaining the baseline model accuracy, a draw back of prior works.

## 7.3 AI Hardware Accelerators

DRAM access requires approximately 100–200× more power than local on-chip cache access [58]. Therefore, currently proposed DNN accelerator architectures propose various schemes to decrease memory footprint and bandwidth. One solution is to keep only a subset of intermediate feature maps at the expense of recomputing convolutions [24]. The presented fusion approach seems to be oversimplified but effective due to high memory access cost. Channeleon is complementary to this work but proposes to keep only compressed feature maps with minimum additional computations. Another work [193] exploits model and feature map sparsity using a more efficient encoding for zeros. While this approach targets similar goals, it requires high sparsity, which is often unavailable in the first and the largest feature maps. In addition, a special control and encoding-decoding logic decrease the benefits of this approach. In our work, compressed feature maps are stored in a dense form without the need of special control and encoding-decoding logic.

cDMA [206] exploit the sparsity inherent in the offload data (from GPU to CPU) to reduces the size of the data structures that are targeted for CPU-side allocations. To overcome the GPU memory capacity bottleneck of DNN training, prior work proposed to virtualize the memory usage of DNNs (vDNN) so that ML researchers can train larger and deeper neural networks beyond what is afforded by the physical limits of GPU memory [205]. By copying GPU-side memory allocations in and out of CPU memory via

the PCIe link, vDNN exposes both CPU and GPU memory concurrently for memory allocations which improves user productivity and flexibility in studying DNN algorithms.

Eager Pruning [278], a sparse training accelerator, works by starting with a dense network and very gradually pruning the lowest-magnitude weights, with fewer than 1% of weights removed every tens of thousands of training iterations; maintaining accuracy limits pruning to comparatively low factors of 1.5–3.5×. The accelerator uses a weight-stationary dataflow where denser filters are distributed over more PEs than sparser filters; to manage the resulting irregularity in collecting partial sums, the authors propose a module that connects the PEs and can either accumulate or route partial sums. Although the Eager Pruning algorithm relies on sorting weights, this does not appear to be considered in the hardware or the latency and energy measurements. In contrast, Procrustes achieves higher pruning factors, does not rely on sorting weights, and avoids the need for a complex interconnect via a novel load balanced dataflow.

TensorDash [167] is a sparse training accelerator that exploits naturally occurring sparsity during training, which appears predominantly in the activations and the gradients: MACs with zero as one of the operands. It borrows upon the sparse-interconnect approach used by Bit-Tactical's front-end [69] and adapt it so that it can be used during training. This interconnect does not sparsify the models, therefore TensorDash needs to add ideas from training acceleration that leverage weight sparsity to extracts additional benefits; it borrows the method in [73] to add weight pruning during the training which is based on sorting weights and as we described earlier, is not efficient for hardware implementation. TensorDash achieves only 1.95× speedups and 1.6× energy savings on pruned models with 90% sparsity.

All other sparse accelerators only support inference. EIE [105] and CambriconX [279] use variants of the compressed sparse column format, which prevents them from efficiently accessing weights during the backward pass. parashar2017scnn [193] and SparTen [93] use an input-stationary dataflow to enable both weight and activation sparsity; however, both use a CSC-like format to encode sparse weights, and neither can be used to accelerate training.

# Chapter 8

# Discussion and Future Work

In this chapter, we conclude the dissertation and discuss potential directions for future research.

## 8.1 Conclusions

In the past decade, on-chip memory capacities have not kept up with the tremendous growth in how much data is collected and stored: in the general-purpose computing systems, CPU last-level caches have stayed around 1MB per core (up to 8MB per core). Similarly, in the special-purpose computing systems used to accelerate applications including DNNs, there is only up to 10s of Megabytes of on-chip memory available. In this context, increasing effective capacity by compressing on-chip memory contents rather than directly increasing memory sizes can reduce the costly off-chip accesses without incurring the costs of a larger silicon area.

Throughout this dissertation, we demonstrate that in order to develop efficient in-hardware compression mechanisms, it is essential to go beyond compressing values in isolation or only a few consecutive data blocks; we can make better compression decisions (which data blocks can be compressed together, what values can be dropped) based on the relevant data points in the entire set of available on-chip data.

In chapter 1, we argue that it is essential to consider both the intra-line and inter-line data redundancy for improving the compression ratio in cache memories as various workloads have different types of redundancy. We propose 2DCC that goes beyond a single type of redundancy and improves the compression ratio on the best performing state-of-the-art method by 1.42×. We further analyzed its applicability to approximate data and showed not only that our proposal improves the compression of exact data, but also that it enables data approximation to achieve competitive compression ratios.

In chapter 2, we discuss a novel method to reduce data redundancy in on-chip cache memory that relies on data similarity. We observed that many on-chip data are similar to each other and therefore can be used to form

clusters, and these clusters are often quite distinct. We proposed Thesaurus, a method to dynamically cluster the data as they come into the memory and then perform delta compression on data within each cluster to avoid storing the redundant chunks of the data. Thesaurus achieves a considerable compression ratio of 2.25× on an extensive set of CPU benchmarks.

In chapter 3, as the first attempt to demonstrate our insights in scratchpad memories, we attacked the problem of activation map compression in DNNs through dynamic clustering. We observed that many activation channels share similar statistics. Thus, we propose Channeleon, a method that looks at the entire set of channels at each layer, clusters them based on activation statistics and performs quantization on each cluster, separately. Compressing in channel-group-wise fashion, rather than compressing each activation channel individually or all layer activations at once as it is done conventionally, enables the values to be compressed to low-bit-widths while retaining the network accuracy in acceptable ranges. We also observed that the distribution of the activation values is often non-uniform and heavily asymmetrical and propose to use an optimized K-Means clustering algorithm as the non-uniform quantizer to further reduce the accuracy loss of low bit-width quantizations.

In chapter 4, we propose an AI hardware training accelerator that produces compressed sparse weights from scratch to reduce the weight footprint by an order of magnitude. In order to compress weights, the compression algorithm looks at the entire set of gradient values and keeps the ones with the highest change. In order to make this technique hardware-friendly, we use a computationally simple quantile estimation method to track weights, and also leverage a novel data-flow and load-balancing scheme that converts sparsity into up to 4× speedups.

## 8.2 Future Research Directions

This section discusses directions for potential future research based on the work performed in this dissertation. While this dissertation focuses on on-chip data compression of cache memories in CPUs and scratchpads in AI accelerators, the techniques and main insights of this dissertation are not limited to these devices.

### 8.2.1 A Unified and Dynamic Compressed Cache

A much more open-ended research direction would be to explore new ways to structure cache memories. The emphasis of this dissertation in chapter 3

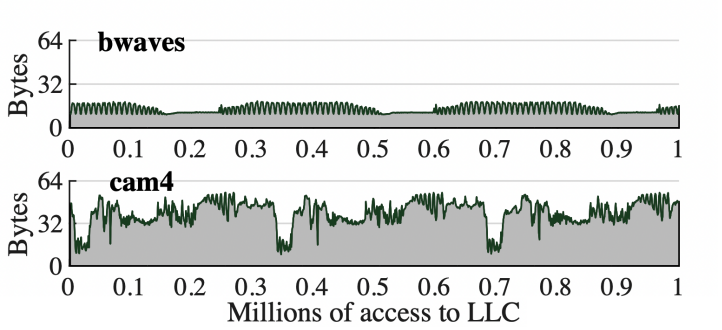



Figure 8.1: Indication of how compressibility varies over time. Y-axis shows the number of unique bytes at each cacheline using Thesaurus compression. 1 million cache insertions after skipping the first 40B instructions.

and chapter 4 is to make the cache compression practical for the existing caches with minimal changed to the existing structures. But, if one can afford to completely redesign how caches are made, there are bigger potential and room to optimize the caches.

To better understand the missed opportunity in current cache designs, let's revisit the observations described in chapter 3 and chapter 4:

- Different workloads have different compression ratios meaning that sizing the structures based on one application for other applications leads to sub-optimal designs.
- Each workload has various phases with different compressiblilty behaviour (see Figure 8.1). This also indicates that during the runtime of a single workload, the ratio of the data to the tag entries and amount of the storage savings changes considerably.

Existing methods size the cache structures (i.e., tag array and data array) based on the average working set footprint over all workloads. Picking this point, will divide the applications into two general groups: tag-bounded and data-bounded; tag-bound applications are ones that can compress data to more than the chosen compression ratio, and data-bounded are the ones that are less compressible.

For example, given a set of workloads, if we observe that the data can be compressed by 2× on average, conventionally we choose to have 2× tag entries compared to data entries. This makes all the workloads with the compression ratio less than 2× to become tag-bounded meaning that cannot compress to more than 2× although there is empty space available in

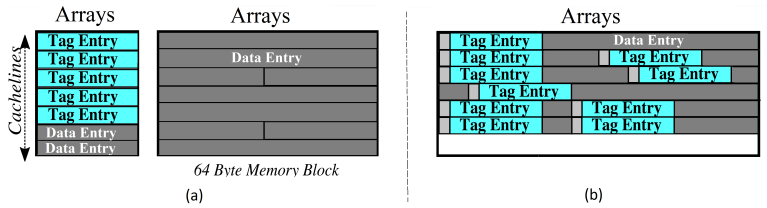



Figure 8.2: Possible unified cache scheme. (a) a unified scheme where the tag/data entry is not limited to be stored only on the tag/data arrays. (b) a unified scheme where the tags and data are stored within a single structure. The light gray box indicates the metadata used for jumping over the data blocks.

the data array. Similarly, the workloads with less compressibility (i.e., uncompressible data) will use up all the entries in the data array and leave some space in tag array structure unused. These inefficiency in using the all available space motivates a better silicon area division.

Now, consider a cache where the tags and data are not limited to only be stored on their corresponding structures. Essentially, in this case, the cache will adapt to the workload and dynamically decide how much of the memory is taken by tags and how much by data. One way to achieve this is by letting the data/tags to be stored on the other structure whenever needed assuming some overhead (e.g., one more bits to indicate which structure that tag/data entry is stored at). We illustrate such cache organization in Figure 8.2(a), however, this complicates the replacement policies, as now a tag insertion may result in a data eviction (and perhaps the eviction of the corresponding tag) and vice versa.

As another possibility, consider a unified structure where the tags and data are stored along each other and there is no physical or virtual division on where the tags and data should be stored. DICE [269] proposes a similar idea to increase the bandwidth of DRAM cache. Essentially, there will be no more tag/data separation when storing each data but rather a bigger block of data where the data and meta data are stored along each other. For example, the first few bits will always be the size of this entry, the next bits store the address/tag bits and the following bytes will be the compressed/uncompressed data block. This cache organization is illustrated in Figure 8.2(b). As the size of each block is known, a data lookup is just a search for the relevant address in the metadata bits of each entry and jump over the data bits until it reaches to the matching address.

One challenge might be the energy implications of such designs. In con-

ventional caches, usually each address lookup results in a tag lookup, and if the tag is located (tag hit) there will be another data lookup. If both the tag and data are stored in the same structure, no matter if a lookup is hit of miss, the entire cache line needs to be accessed. This might result in more energy if the tag does not exists in the cache.

Obviously, there are many other design and algorithmic choices to make here such as whether to align the blocks or dedicate some part of the memory to store some pointers for faster search which needs careful study and evaluation.

### 8.2.2 Compressing Across All Levels of the Memory Hierarchy

Chapter 3 and chapter 4 describe in-hardware mechanisms and evaluate them on the last-level caches in CPUs. An interesting research direction would be to understand how to apply those techniques to efficiently compress the data across all levels of the memory hierarchy, from data origin to where the data is being consumed.

Whole system compression could reduce the overheads of compression and decompression at every level of memory hierarchy as now a block can be transferred in a compressed format through various levels. Furthermore, as each component plays a slightly different role while storing the data, different applications with various memory access pattern will benefit from them differently. For example, cache memories do well with temporal locality as they cache the most frequently used blocks. Cache compression will have more benefit with the applications with greater temporal locality, whereas main memory compression might be helpful for the benchmarks with spatial locality; main memory compression can reduce the bandwidth on streaming patterns.

Hence, if the goal is to improve performance of a wide variety of applications in general-purpose computers, employing the compression across all levels of the hierarchy can bring the best results.

Below, we list the benefits that compressing each part of the memory system adds:

**Compressing all cache levels**: LLC compression improves performance by reducing the number of main memory accesses, however, there still is considerable data movement among the different levels of the cache. Compressing the data in smaller and more latency critical caches like L1 cache can be beneficial in reducing those data movements and needs special considerations.

The methods we proposed in Chapter 2 and 3 can be used throughout the entire cache hierarchy due to its very low decompression latency, but things such as clustering parameters can be different for each cache. If different compression/decompression process at each step, results in slightly different format for the compressed block, there is a need for a fast format conversion mechanism.

There is an interesting research topic here to study this trade-off as well as figuring out where to put these compression decompression units, whether to convert the compression format on the fly or after the data is stored at each level, and so on.

**Compressing the bus between LLC and main memory**: There are two major benefits in compressing the link between the caches and main memory. First, it will reduce the the average latency of memory accesses if both main memory and cache are compressed. Second it can decrease the bus energy consumption as there will be less data (i.e., fewer bits per same information) being transferred. Cache compression mechanisms reduce the main memory bandwidth by reducing the LLC misses and therefore main memory accesses due to the misses, but they cannot reduce the bandwidth required to transfer this missed block from main memory to the LLC. As we mentioned in chapter 2, delta compression methods are widely used to compress streams. The method we propose in Chapter 2, reaches to delta compressed blocks in more efficient way. Therefore, we believe our compression format will be efficient for data movement in the bus as well. The compression scheme of Thesaurus can be simplified to only consider multiple blocks within an specific range (e.g., some number of consecutive blocks) while performing the compression.

Compressing the traffic on the bus between LLC and main memory can be especially beneficial for memory-bandwidth-intensive applications (i.e., applications where main memory bandwidth is the bottleneck for performance).

**Compressing main memory**: Main memory compression can benefit the system performance in couple ways: it can reduce high latency disk accesses, and, it can enable less main memory bus contention by reducing the memory bandwidth requirement.

Methods described in Chapter 2 and 3 can be leveraged for the entire cache hierarchy compression. For the data being communicated on the bus methods such as Huffman can be a good candidate. Lastly, for the memory compression proposals like [120, 268] can be leveraged.

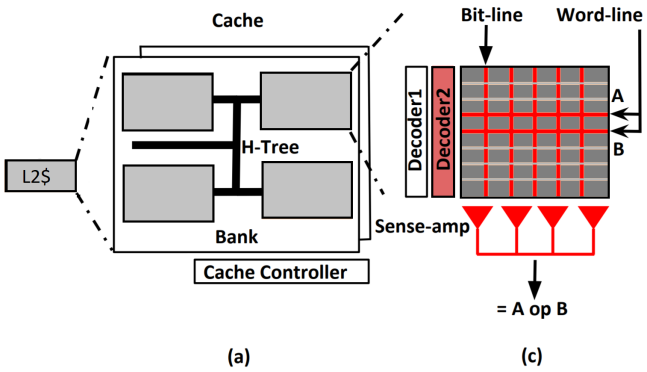



Figure 8.3: Compute Cache overview from [18]. (a) Cache Geometry (b) In-place compute in a sub-array

### 8.2.3 Compressed Compute Caches

Processing-in-memory methods reduces the overheads of moving the data from where it is stored to where the computation happens [277]. The work in [18] proposes Compute Cache, an architecture that enables in-place computation in caches. Compute Caches uses emerging bit-line SRAM circuit technology to transforms existing caches into very large vector computational units. In these architectures, several operations such as logical AND, XOR, etc. can be performed on the cached data by simultaneously activating multiple word-lines and sensing the resulting voltage over the shared bit-lines.

Neural Cache [77] further augments compute capability to efficiently support fixed point arithmetic operations. It re-purposes the SRAM arrays in LLC as bit-serial ALUs, and builds an accelerator for CNN inference. Therefore, neural cache act as a massively parallel compute units capable of running inferences for Deep Neural Networks. The proposed architecture also supports quantization in-cache. Later, Duality Cache [85] extends the operations to support the floating-point arithmetic functions to enable general-purpose data parallel applications to run on caches.

These proposals are beneficial in eliminating the explicit data transfer through PCIe from host to device memory; which can be a barrier to achieve speedups. For example, the initial data transfer between the host (i.e., CPU) and device memory (e.g., GPU) is costly especially when data reuse is not high.

The efficiency of Compute Caches arises from two main sources: *reduced*

*data movement* and *massive parallelism*. Compute Caches reduce the data movement by a) reducing the transfer between caches and the cores and different levels of hierarchy, as well as, b) reducing transfer between a cache sub-array to its controller through in-cache interconnect. In order to make parallelism, one challenge that these proposals have to overcome is to align the data (with the help of compiler) so that data can be computed in bit-aligned fashion.

**Compression benefits**: Aside from the obvious benefit of compressing Compute Caches, which is being able to store more data, compression these caches can make the data movement even less expensive. A study can look into the gains and overheads of compressing the cache and memories in the memory hierarchy, as well as, the interconnect.

Compression can make Compute Cache operations cheaper as well. The two types of operation that takes place in Compute Caches are *in-place* operations, where the data is manipulated in the structures itself, and *near-place* operations where the source operands are read out from the cache sub-arrays and the operation is performed in a logic unit placed close to the cache controller [18]. While the benefits of the compression can be limited for in-place operations, the benefits for near-place operations can be considerable; as essentially less bit-lines needs to be turned on and transferred around, the cost of reading data out from structures can be reduced for applications requiring considerable amount of near-place operations.

### 8.2.4 Activation Map Compression in Training

There is plenty of efficient inference techniques that reduce the number of trainable parameters and the computation FLOPs [46, 106, 216], however, parameter-efficient techniques do not directly save the training memory [47].

While chapter 5 evaluates activation map compression mechanism in the inference phase of DNNs, an immediate follow up work would be to investigate how to efficiently apply that to the DNN training phase. It is essential to reduce the size of intermediate activations required by back-propagation, which is the key memory bottleneck for efficient on-device training [47].

The benefits from reducing the activation map memory footprint will be significantly more than in the training phase: there is more data that needs to be transferred around (i.e., activations, gradients, etc.) and stored for a longer time to be reused in the backward step.

Moreover, training tasks are usually done on batches of images, rather than single image. This potentially help compression to find more similar

chunks of data to cluster and bring more benefit to the training phase.

One thing that is different in the training phase as compared to the inference phase is the behaviour of batch-norm layer [128]. Batch-norm layer is a layer that generally comes after the computation layer. It normalize the batch by first subtracting its mean, then dividing it by its standard deviation. Further scale and shift with the parameters of the batch normalization layer. Once the training has ended, each batch normalization layer possesses a specific set parameters, some being computed using an exponentially weighted average during training [128]. During inference, the batch normalization acts as a simple linear transformation of what comes out of the previous layer, often a convolution layer. As convolution is also a linear transformation, it means that both operations can be merged (folded) into a single linear transformation. This is a common practice to make inference faster and reduce the number of parameters. Therefore, the activation statistics coming out of compute layers in training is different from inference with folded batch-norm (and even fused ReLU), which means they need special considerations such as different clustering hyperparamters. In order to form better clusters, it is also probable that there is a need for normalizing activations of compute layer before compressing them.

Another challenge is that the parameters and therefore the range of activations are constantly changing during the training, especially in the early iterations. This requires that the method capture these dynamically changing activation distribution. While the clustering method proposed in Channeleon should be able to handle this, it might need different clustering parameters for different training epochs, which needs careful investigation. Later epochs won't have this issue as the parameter distribution tends to become stable and the parameters change relatively little compared to the earlier iterations.

The last challenge is the precision requirements of training. While in inference some accuracy can be traded for performance, introducing accuracy errors can be critical in training. After each inference pass, the activations are discarded; in training, on the other hand, they will be used in the back propagation phase to tune the parameters for the next step, and therefore using lossy compression methods might cause the model converge very slowly or not converge at all. One possibility might be to use a different precision to pass the activations to next layer and to store them to be reused in back propagation as suggested by [133]. Then there will be a trade off between the precision and the memory used for each of those paths.

### 8.2.5 Compression-Aware Regularization

In chapter 5, we presented a compression mechanism to reduce the memory footprint of the intermediate activation maps. This scheme does not change the network structure and can work with pretrained models. One interesting research direction here is that how this compression can be boosted by knowing the training time information; in other words, developing an efficient compression aware training algorithm.

Currently, the convention to make models more compact in the training time is to make them sparse, i.e., bring unimportant values down to zero; an example of such mechanism is illustrated in Figure 8.4(a) where values below some threshold (boxes with lighter yellow and pink colors in the figure) are pushed towards or replaced with zero (the white boxes in the figure). This can be achieved by using terms such as L1-norm as in [87]. They propose to aid training of neural networks by explicitly encoding in the cost function to be minimized, the desire to achieve sparser activation maps.

Although sparse activations are more compressible, zeroing out values will only be beneficial in a specific compression mechanism. Moreover, not all the networks will tolerate the error introduce by ignoring the connections [264]. A more generic way of doing this is to reduce the entropy in the values.

Therefore, we can propose a compression mechanism that goes beyond value sparsification and uses a regularization term that produce more compressible data, i.e., more repeated data. Such a regularization term can push the activation maps to have identical value if they are similar to each other. A possible way to accomplish this is by tiling the activation map tensor into many chunks, clustering these chunks, and replacing the items within each cluster with the their corresponding clusteroid. Therefore, the cost function can be defined as the difference between these items in each chunk, from the corresponding clusteroid.

Figure 8.4(b) illustrates the case where $3 \times 3$ tiles of data are clustered into 5 clusters based on the pixel similarity. These clusters are distinguished by different colors (black, blue, red, green and maroon boxes, each with 3,2,2,1, and 1 members respectively), and the clusteroid for each of those clusters (five $3 \times 3$ tiles) are depicted on top and right side of the figure. For example, it is visible on the bottom of this figure that the tiles in the first cluster with 3 members can be replaced by a single clusteroid tile; this tile repetition enables a potential of 3 times compression on this cluster.

We can introduce a regularization term that encourages the values to be similar with the following cost function using L1-norm:

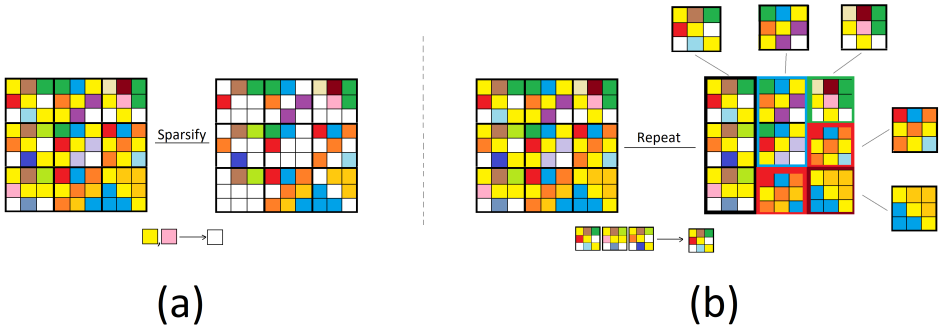



Figure 8.4: Activation map compression. (a) a sparsification method used in conventional compression methods. (b) possible compression method where similar chunks of activations are pushed to produce same values.

$$cost = \sum_{i=0}^{k} \sum_{j=0}^{n} |chunk_{ij} - centroid_i| \tag{8.1}$$

where $k$ indicates the total number of clusters(chunks), $n$ indicates the total number of items(tiles) in each cluster and $centroid_i$ is the centroid chosen for that cluster.

The clustering task can be efficiently performed following the method proposed in chapter 4. The clusteroids can be the average of the members of each cluster, or simply the first member in each cluster, e.g., the first tile that maps to an specific hash in an LSH-based clustering method. Since each training iteration, forms the clutters dynamically, clusters repeatedly get chances to better capture the similarity of the tiles if they are not perfectly formed in an earlier iteration.

It is an interesting research question to see how much this regularization term helps the compression and what will be the accuracy compression trade-off as the values are being altered.

# Bibliography


[1]

[2] DEFLATE Compressed Data Format Specification version 1.3. datatracker.ietf.org/doc/html/rfc1951.

[3] Gnu gzip compression format. www.gnu.org/software/gzip/.

[4] Intel® core™ i5-750 processor specifications, howpublished = https://www.intel.com/content/www/us/en/products/sku/42915/intel-core-i5750-processor-8m-cache-2-66-ghz/specifications.html.

[5] Intel® Sunny Cove microarchitecture. https://en.wikichip.org/wiki/intel/microarchitectures/nehalem_(client).

[6] Intel® sunny cove microarchitecture.

[7] Microsoft windows bitmap format (bmp), version 5. www.loc.gov/preservation/digital/formats/fdd/fdd000189.shtml. 2011.

[8] rsync. rsync.samba.org/.

[9] Truevision tga file format, version 2.0. www.loc.gov/preservation/digital/formats/fdd/fdd000180.shtml. 2005.

[10] zip compression format specifications. pkware.cachefly.net/webdocs/APPNOTE/APPNOTE-6.3.9.TXT.

[11] zlib compression format, version 1.2.11. zlib.net/. 2017.

[12] First the Tick, Now the Tock: Next Generation Intel® Microarchitecture (Nehalem). https://www.intel.com/pressroom/archive/reference/whitepaper_Nehalem.pdf, 2008.

[13] International technology roadmap for semiconductors (itrs). 2011.

[14] SPEC releases major new CPU benchmark suite. 2017.

[15] Bulent Abali, Hubretus Franke, Dan Poff, R Saccone, Charles O. Schulz, Lorraine M. Herger, and T.Basil Smith. Memory expansion technology (mxt): Software support and performance. *IBM JRD*, 2001.

[16] Mohamed S. Abdelfattah, Andrei Hagiescu, and Deshanand Singh. Gzip on a chip: High performance lossless data compression on fpgas using opencl. In *IOWCL*, 2014.

[17] Dimitris Achlioptas. Database-friendly Random Projections. In *SPDS*, 2001.

[18] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das. Compute caches. In *HPCA*, 2017.

[19] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. Extremely large minibatch SGD: training ResNet-50 on ImageNet in 15 minutes. *DLSS NeurIPS workshop*, 2017.

[20] A. R. Alameldeen and D. A. Wood. Adaptive cache compression for high-performance processors. In *ISCA*, 2004.

[21] Alaa R. Alameldeen and David Wood. Frequent pattern compression: A significance-based compression scheme for L2 caches. Technical report, University of Wisconsin-Madison, 2004.

[22] Jorge Albericio, Alberto Delmás, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. Bit-pragmatic deep neural network computing. In *MICRO-50*, 2017.

[23] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: ineffectual-neuron-free deep neural network computing. In *ISCA*, 2016.

[24] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-layer CNN accelerators. In *MICRO*, 2016.

[25] Angelos Arelakis, Fredrik Dahlgren, and Per Stenström. HyComp: A Hybrid Cache Compression Method for Selection of Data-type-specific Compression Methods. In *MICRO*, 2015.

[26] Angelos Arelakis and Per Stenström. $SC^2$: A Statistical Compression Cache Scheme. In *ISCA*, 2014.

[27] Raman Arora, Amitabh Basu, Poorya Mianjy, and Anirbit Mukherjee. Understanding deep neural networks with rectified linear units. In *ICLR*, 2018.

[28] Kaushik Balasubramanian Kai Cheng Prashant Damle Sham Datta Chet Douglas Kenneth Gibson Benjamin Graniello John Grooms Naga Gurumoorthy Ivan Cuevas Escareno Tiffany Kasanicky Kunal Khochare Zhiming Li Sreenivas Mandava Rick Mangold Sai Muralidhara Shamima Najnin Bill Nale Jay Pickett Shekoufeh Qawami Tuan Quach Bruce Querbach Camille Raad Andy Rudoff Ryan Saffores Ian Steiner Muthukumar Swaminathan Shachi Thakkar Vish Viswanathan Dennis Wu Cheng Xu Asher Altman, Mohamed Arafa. Intel® Optane™ Data Center Persistent Memory. In *HotChips*, 2019.

[29] Stephane Ayache, Ronan Sicre, and Thierry Artières. Transfer learning by weighting convolution. In *IJCNN*, 2020.

[30] Mehmet Aygün, Yusuf Aytar, and Hazým Kemal Ekenel. Exploiting convolution filter patterns for transfer learning. In *ICCVW*, 2017.

[31] Seungcheol Baek, Hyung Gyu Lee, Cyrysostomos Nicopoulos, Junghee Lee, and Jongma Kim. Ecm: Effective capacity maximizer for high-performance compressed caching. In *HPCA*, 2013.

[32] Ron Banner, Yury Nahshan, Elad Hoffer, and Daniel Soudry. Post-training 4-bit quantization of convolution networks for rapid-deployment, 2019.

[33] Chaim Baskin, Natan Liss, Eli Schwartz, Evgenii Zheltonozhskii, Raja Giryes, Alex M Bronstein, and Avi Mendelson. Uniq: Uniform noise injection for non-uniform quantization of neural networks. *TOCS*, 2021.

[34] S. Beamer, K. Asanović, and D. Patterson. The GAP benchmark suite. *arXiv:1508.03619*, 2015.

[35] Scott Beamer III. *Understanding and improving graph algorithm performance*. PhD thesis, University of California, Berkeley, 2016.

[36] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*. 2012.

[37] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

[38] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*, 2008.

[39] J. Bonwick and B. Moore. Zfs: The last word in file systems. 2007.

[40] Jalil Boukhobza, Stéphane Rubini, Renhai Chen, and Zili Shao. Emerging NVM: A Survey on Architectural Integration and Research Challenges. *TODAES*, 2017.

[41] Andrew Brock, Soham De, Samuel L Smith, and Karen Simonyan. High-performance large-scale image recognition without normalization. *arXiv preprint arXiv:2102.06171*, 2021.

[42] Randal Bryant. Data-intensive supercomputing: The case for disc. 2007.

[43] James Bucek, Klaus-Dieter Lange, and J'oakim von Kistowski. SPEC CPU2017 — Next-generation Compute Benchmark. In *ICPE*, 2018.

[44] Jeremy Buhler. Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics*, 2001.

[45] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *SPAA*, 2009.

[46] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-all: Train one network and specialize it for efficient deployment. In *ICLR*, 2020.

[47] Han Cai, Chuang Gan, Ligeng Zhu, and Song Han. Tinytl: Reduce activations, not trainable parameters for efficient on-device learning. In *NeurIPS*, 2020.

[48] Yaohui Cai, Zhewei Yao, Zhen Dong, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. ZeroQ: A Novel Zero Shot Quantization Framework. In *CVPR*, 2020.

[49] Yu Cai, Erich F. Haratsch, Onur Mutlu, and Ken Mai. Threshold voltage distribution in mlc nand flash memory: Characterization, analysis, and modeling. In *DATE*, 2013.

[50] Yu Cai, Gulay Yalcin, Onur Mutlu, Erich F. Haratsch, Adrian Cristal, Osman S. Unsal, and Ken Mai. Flash correct-and-refresh: Retention-aware error management for increased flash memory lifetime. In *ICCD*, 2012.

[51] Daniel Rodrigues Carvalho and André Seznec. Understanding cache compression. *TACO*, 2021.

[52] Jingfei Chang, Yang Lu, Ping Xue, Yiqun Xu, and Zhen Wei. ACP: Automatic Channel Pruning via Clustering and Swarm Intelligence Optimization for CNN. In *ArXiv*, 2021.

[53] Moses S. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, 2002.

[54] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ASPLOS*, 2014.

[55] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. In *ICML*, 2015.

[56] Xi Chen, Lei Yang, Robert P. Dick, Li Shang, and Haris Lekatsas. C-Pack: A High-Performance Microprocessor Cache Compression Algorithm. *IEEE Transactions on Very Large Scale Integration Systems*, 2010.

[57] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *ISCA*, 2016.

[58] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2019.

[59] David Cheriton, Amin Firoozshahian, Alex Solomatnikov, John P. Stevenson, and Omid Azizi. Hicamp: Architectural support for efficient concurrency-safe shared structured data access. *SIGPLAN Not.*, March 2012.

[60] Yu-Der Chih, Yi-Chun Shih, Chia-Fu Lee, Yen-An Chang, Po-Hao Lee, Hon-Jarn Lin, Yu-Lin Chen, Chieh-Pu Lo, Meng-Chun Shih, Kuei-Hung Shen, Harry Chuang, and Tsung-Yung Jonathan Chang. 13.3 a 22nm 32mb embedded stt-mram with 10ns read speed, 1m cycle write endurance, 10 years retention at 150°c and high immunity to magnetic field interference. In *ISSCC*, 2020.

[61] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. Pact: Parameterized clipping activation for quantized neural networks. In *ICLR*, 2018.

[62] Yoni Choukroun, Eli Kravchik, Fan Yang, and Pavel Kisilev. Low-bit quantization of neural networks for efficient inference. In *ICCV workshop*, 2019.

[63] Esha Choukse, Mattan Erez, and Alaa Alameldeen. Compresspoints: An evaluation methodology for compressed memory systems. *IEEE CAL*, 2018.

[64] Esha Choukse, Michael B. Sullivan, Mike O'Connor, Mattan Erez, Jeff Pool, David Nellans, and Stephen W. Keckler. Buddy Compression: Enabling Larger Memory for Deep Learning and HPC Workloads on GPUs. In *ISCA*, 2020.

[65] Ondrej Chum, James Philbin, and Andrew Zisserman. Near duplicate image detection: min-hash and tf-idf weighting. In *BMVC*, 2008.

[66] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *JMLR*, 2011.

[67] Yann Le Cun, John S. Denker, and Sara A. Solla. Optimal brain damage. In David S. Touretzky, editor, *NeurIPS*, 1990.

[68] J. D. Deaton and A. Bacon. White paper: Smashing big data costs with gzip hardware. `http://www.aha.com/Uploads/GZIPBenefitsWhitepaper11.pdf`, 2015.

[69] Alberto Delmas Lascorz, Patrick Judd, Dylan Malone Stuart, Zissis Poulos, Mostafa Mahmoud, Sayeh Sharify, Milos Nikolic, Kevin Siu, and Andreas Moshovos. Bit-tactical: A software/hardware approach to exploiting value and bit sparsity in neural networks. In *ASPLOS*, 2019.

[70] Timothy E. Denehy, Windsor W. Hsu, Timothy E. Denehy, and Windsor W. Hsu. Duplicate management for reference data. In *IBM Research Report RJ10305*, 2003.

[71] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR*. Ieee, 2009.

[72] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *NeurIPS*, 2014.

[73] Tim Dettmers and Luke Zettlemoyer. Sparse networks from scratch: Faster training without losing performance. *arXiv preprint arXiv:1907.04840*, 2019.

[74] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. *NAACL*, 2019.

[75] Qing Dong, Zhehong Wang, Jongyup Lim, Yiqun Zhang, Yi-Chun Shih, Yu-Der Chih, Jonathan Chang, David Blaauw, and Dennis Sylvester. A 1mb 28nm stt-mram with 2.8ns read access time at 1.2v vdd using single-cap offset-cancelled sense amplifier and in-situ self-write-termination. In *ISSCC*, 2018.

[76] Julien Dusser, Thomas Piquet, and André Seznec. Zero-content augmented caches. In *ICS*, 2009.

[77] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaaauw, and Reetuparna Das. Neural cache: Bit-serial in-cache acceleration of deep neural networks. *ISCA*, Jun 2018.

[78] Magnus Ekman and Per Stenström. A Robust Main-Memory Compression Scheme. In *ISCA*, 2005.

[79] Ali Mamdouh Elkahky, Yang Song, and Xiaodong He. A multi-view deep learning approach for cross domain user modeling in recommendation systems. In *WWW*, 2015.

[80] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, 1996.

[81] Jun Fang, Ali Shafiee, Hamzah Abdel-Aziz, David Thorsley, Georgios Georgiadis, and Joseph H Hassoun. Post-training piecewise linear quantization for deep neural networks. In *ECCV*, 2020.

[82] Sean Fox, Stephen Tridgell, Craig Jin, and Philip H. W. Leong. Random projections for scaling machine learning on FPGAs. In *FPT*, 2016.

[83] Peter Frankl and Hiroshi Maehara. The Johnson-Lindenstrauss lemma and the sphericity of some graphs. *Journal of Combinatorial Theory, Series B*, 1988.

[84] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *ICLR*, 2019.

[85] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. Duality cache for data parallel acceleration. In *ISCA*, 2019.

[86] Jayesh Gaur, Alaa R. Alameldeen, and Sreenivas Subramoney. Base-victim compression: An opportunistic cache compression architecture. In *ISCA*, 2016.

[87] Georgios Georgiadis. Accelerating convolutional neural networks via activation map compression. In *CVPR*, 2019.

[88] Amin Ghasemazar, Mohammad Ewais, Prashant Nair, and Mieszko Lis. 2DCC: Cache Compression in Two Dimensions. In *DATE 2020*, 2020.

[89] Mohammad Ghayoumi, Miguel Gomez, Kate E. Baumstein, Narindra Persaud, and Andrew J. Perlowin. Local Sensitive Hashing (LSH) and Convolutional Neural Networks (CNNs) for Object Recognition. In *ICMLA*, 2018.

[90] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity Search in High Dimensions via Hashing. In *VLDB*, 1999.

[91] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*, 2010.

[92] Maximilian Golub, Guy Lemieux, and Mieszko Lis. Full Deep Neural Network Training on a Pruned Weight Budget. In *SysML*, 2019.

[93] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and TN Vijaykumar. SparTen: A Sparse Tensor Accelerator for Convolutional Neural Networks. In *MICRO*, 2019.

[94] Jiong Gong, Haihao Shen, Guoming Zhang, Xiaoli Liu, Shane Li, Ge Jin, Niharika Maheshwari, Evarist Fomenko, and Eden Segal. Highly efficient 8-bit low precision inference of convolutional neural networks with intelcaffe. In *ReQuEST@ASPLOS*, 2018.

[95] Ruihao Gong, Xianglong Liu, Shenghu Jiang, Tianxiang Li, Peng Hu, Jiazhen Lin, Fengwei Yu, and Junjie Yan. Differentiable soft quantization: Bridging full-precision and low-bit neural networks. In *CVPR*, 2019.

[96] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.

[97] Google. Compcache. `https://code.google.com/archive/p/compcache/`, 2015.

[98] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Łukasz Wesołowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv:1706.02677*, 2017.

[99] Alex Graves and Jüergen Schmidhuber. Framewise phoneme classification with bidirectional lstm networks. In *IJCNN*, 2005.

[100] Denis Gudovskiy, Alec Hodgkinson, and Luca Rigazio. DNN feature map compression using learned representation over GF (2). In *ECCV Workshops*, 2018.

[101] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic Network Surgery for Efficient DNNs. In *NeurIPS*, 2016.

[102] Yunhui Guo, Honghui Shi, Abhishek Kumar, Kristen Grauman, Tajana Rosing, and Rogerio Feris. Spottune: Transfer learning through adaptive fine-tuning. In *CVPR*, 2019.

[103] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *PMLR*, 2015.

[104] E. G. Hallnor and Steven K. Reinhardt. A unified compressed memory hierarchy. In *HPCA*, 2005.

[105] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. EIE: efficient inference engine on compressed deep neural network. In *ISCA*, 2016.

[106] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *ICLR*, 2016.

[107] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. In *NeurIPS*, 2015.

[108] Stephen José Hanson and Lorien Pratt. Advances in neural information processing systems 1. chapter Comparing Biases for Minimal Network Construction with Back-propagation. 1989.

[109] Babak Hassibi and David G Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *NeurIPS*, 1993.

[110] Babak Hassibi, David G. Stork, and Greg J. Wolff. Optimal brain surgeon and general network pruning. In *ICNN*, 1993.

[111] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on Imagenet classification. In *ICCV*, 2015.

[112] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[113] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *ICCV*, 2017.

[114] John L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News*, September 2006.

[115] Byeongho Heo, Jeesoo Kim, Sangdoo Yun, Hyojin Park, Nojun Kwak, and J. Choi. A comprehensive overhaul of feature distillation. *ICCV*, 2019.

[116] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. In *NeurIPS*, 2014.

[117] Bo Hong, Demyn Plantenberg, Darrell D. E. Long, and Miriam Sivan-Zimet. Duplicate Data Elimination in a SAN File System. In *MSST*, 2004.

[118] Cheeun Hong, Heewon Kim, Junghun Oh, and Kyoung Mu Lee. DAQ: Distribution-Aware Quantization for Deep Image Super-Resolution Networks. *arXiv preprint arXiv:2012.11230*, 2020.

[119] Seokin Hong, Bulent Abali, Alper Buyuktosunoglu, Michael B. Healy, and Prashant J. Nair. Touché: Towards ideal and efficient cache compression by mitigating tag area overheads. In *MICRO*, 2019.

[120] Seokin Hong, Prashant J. Nair, Bulent Abali, Alper Buyuktosunoglu, Kyu-Hyoun Kim, and Michael Healy. Attaché: Towards ideal memory compression by mitigating metadata bandwidth overheads. In *MICRO*, 2018.

[121] Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). In *ISSCC*, 2014.

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

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

[124] Zehao Huang and Naiyan Wang. Data-driven sparse structure selection for deep neural networks. In *ECCV*, 2018.

[125] Itay Hubara, Daniel Soudry, and Ran El Yaniv. Binarized neural networks. In *NeurIPS*, 2016.

[126] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 1952.

[127] Piotr Indyk and Rajeev Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *STC*, 1998.

[128] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *ICML*, 2015.

[129] Mafijul Md Islam and Per Stenström. Characterization and Exploitation of Narrow-width Loads: The Narrow-width Cache Approach. In *CASES*, 2010.

[130] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module, 2019.

[131] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *CVPR*, 2018.

[132] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866*, 2014.

[133] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. Gist: Efficient data encoding for deep neural network training. In *ISCA*, 2018.

[134] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). 2010.

[135] Yifei Jiang, Yi Li, Yiming Sun, Jiaxin Wang, and David P. Woodruff. Single pass entrywise-transformed low rank approximation. In *ICML*, 2021.

[136] Yu Jiang. Evaluation of huffman coding in memory compression. Master's thesis, Chalmers University, 2014.

[137] William Johnson and J0rdan Lindenstrauss. Extensions of Lipschitz mappings into a Hilbert space. *CMAP*, 1982.

[138] Norman P. Jouppi, Andrew B. Kahng, Naveen Muralimanohar, and Vaishnav Srinivas. Cacti-io: Cacti with off-chip power-area-timing models. *VLSI*, 2015.

[139] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Bo-

den, Al Borchers, et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *ISCA*, 2017.

[140] Patric Judd, Jorge Albericio, Talor Hetherington, Tor M. Aamodt, and Andreas Moshovos. Stripes: Bit-serial deep neural network computing. In *MICRO*, 2016.

[141] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M. Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Proteus: Exploiting numerical precision variability in deep neural networks. In *ICS*, 2016.

[142] Sang Woo Jun, Kermin E Fleming, Michael Adler, and Joel Emer. Zip-io: Architecture for application-specific compression of big data. In *FPT*, 2012.

[143] Jungrae Kim, Michael Sullivan, Esha Choukse, and Mattan Erez. Bit-plane compression: Transforming data for better compression in many-core architectures. In *ISCA*, 2016.

[144] Min Soo Kim, Alberto Antonio Del Barrio Garcia, Hyunjin Kim, and Nader Bagherzadeh. The effects of approximate multiplication on convolutional neural networks. *TETC*, 2021.

[145] Philip Koopman. *Stack Computers: The New Wave*. Computers and their applications. Emily Horwood, 1989.

[146] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. `https://arxiv.org/abs/1806.08342`, 2018.

[147] Krizhevsky, Alex, Sutskever, Ilya, Hinton, and Geoffrey E. Imagenet classification with deep convolutional neural networks. *NeurIPS*, 2012.

[148] Alex Krizhevsky. Learning multiple layers of features from tiny images. Master's thesis, University of Toronto, 2009.

[149] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. Understanding Reuse, Performance, and Hardware Cost of DNN Dataflow: A Data-Centric Approach. In *MICRO*, 2019.

[150] Alberto Delmás Lascorz, Sayeh Sharify, Isak Edo, Dylan Malone Stuart, Omar Mohamed Awad, Patrick Judd, Mostafa Mahmoud, Milos Nikolic, Kevin Siu, Zissis Poulos, and Andreas Moshovos. Shapeshifter:

Enabling fine-grain data width adaptation in deep learning. In *MICRO*, 2019.

[151] Chris Lattner and Vikram S Adve. Transparent pointer compression for linked data structures. In *MSP*, 2005.

[152] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 2015.

[153] Yann LeCun, John S. Denker, and Sara A. Solla. Optimal brain damage. In *NeurIPS*, 1990.

[154] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*. 2012.

[155] Sungjin Lee, Jihoon Park, Kermin Fleming, Jihong Kim, et al. Improving performance and lifetime of solid-state drives using hardware-accelerated compression. *TOCE*, 2011.

[156] Bing Li, Bonan Yan, and Hai Li. An Overview of In-Memory Processing with Emerging Non-Volatile Memory for Data-Intensive Applications. In *GLSVLSI*, 2019.

[157] Bowen Li, Kai Huang, Siang Chen, Dongliang Xiong, Haitian Jiang, and Luc Claesen. DFQF: data free quantization-aware fine-tuning. In *ACML*, 2020.

[158] Chunyuan Li, Heerad Farkhoor, Rosanne Liu, and Jason Yosinski. Measuring the intrinsic dimension of objective landscapes. In *ICLR*, 2018.

[159] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. In *ICLR*, 2017.

[160] Ping Li, Trevor J. Hastie, and Kenneth W. Church. Very sparse random projections. In *KDD*, 2006.

[161] Rundong Li, Yag Wang, Feng Liang, Hongwei Qin, Junjie Yan, and Rui Fan. Fully quantized network for object detection. In *CVPR*, 2019.

[162] Fan Liang, Wei Yu, Dou An, Qingyu Yang, Xinwen Fu, and Wei Zhao. A survey on big data market: Pricing, trading and protection. *IEEE Access*, 2018.

[163] Wanqi Liu, Hanchen Wang, Ying Zhang, Wei Wang, and Lu Qin. I-lsh: I/o efficient c-approximate nearest neighbor search in high-dimensional space. In *ICDE*, 2019.

[164] Xingchao Liu, Mao Ye, Dengyong Zhou, and Qiang Liu. Post-training quantization with multiple points: Mixed precision without mixed precision. In *AAAI*, 2021.

[165] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, S. Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. *ICCV*, 2017.

[166] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. In *ICCV*, 2017.

[167] Mostafa Mahmoud, Isak Edo, Ali Hadi Zadeh, Omar Mohamed Awad, Gennady Pekhimenko, Jorge Albericio, and Andreas Moshovos. Tensordash: Exploiting sparsity to accelerate deep neural network training. In *MICRO*, 2020.

[168] J Mandelman, Robert H. Dennard, Gary Bronner, J.k. DeBrosse, Rama Divakaruni, Ying Li, and Carol Radens. Challenges and future directions for the scaling of dynamic random-access memory (dram). *IBM JRD*, 2002.

[169] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. Detecting near-duplicates for web crawling. In *WWW*, 2007.

[170] George Marsaglia. Xorshift rngs. *JSS*, 2003.

[171] Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks. *arXiv preprint arXiv:1804.07612*, 2018.

[172] Hui Miao, Ang Li, Larry Davis, and Amol Deshpande. Towards unified data and lifecycle management for deep learning. In *ICDE*, 2017.

[173] Joshua San Miguel, Jorge Albericio, Andreas Moshovos, and Natalie Enright Jerger. Doppelgänger: a cache for approximate computing. In *MICRO*, 2015.

[174] Maximilian Miller, Chengsheng Zhu, and Yana Bromberg. Clubber: removing the bioinformatics bottleneck in big data analyses. *JIB*, 2017.

[175] Sparsh Mittal and Jeffrey S. Vetter. Ayush: Extending lifetime of sram-nvm way-based hybrid caches using wear-leveling. In *MASCOTS*, 2015.

[176] Decebal Constantin Mocanu, Elena Mocanu, Peter Stone, Phuong H Nguyen, Madeleine Gibescu, and Antonio Liotta. Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature Communications*, 2018.

[177] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440*, 2016.

[178] Tomas Möller. A Fast Triangle-Triangle Intersection Test. *Journal of Graphics Tools*, 1997.

[179] Hesham Mostafa and Xin Wang. Parameter efficient training of deep convolutional neural networks by dynamic sparse reparameterization. In *ICML*, 2019.

[180] Avilash Mukherjee, Kumar Saurav, Prashant Nair, Sudip Shekhar, and Mieszko Lis. A Case for Emerging Memories in DNN Accelerators. In *DATE*, 2021.

[181] Naveen Muralimanohar, Rajeer Balasubramonian, and Norman Jouppi. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *MICRO*, 2007.

[182] Onur Mutlu. Memory scaling: A systems architecture perspective. In *IMW*, 2013.

[183] Markus Nagel, Rana Ali Amjad, Mart Van Baalen, Christos Louizos, and Tijmen Blankevoort. Up or down? adaptive rounding for post-training quantization. In *ICML*, 2020.

[184] Markus Nagel, Mart van Baalen, Tijmen Blankevoort, and Max Welling. Data-free quantization through weight equalization and bias correction. In *ICCV*, 2019.

[185] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *ICML*. Omnipress, 2010.

[186] Seyyed Mahdi Najmabadi, Zhe Wang, Yousef Baroud, and Sven Simon. Online bandwidth reduction using dynamic partial reconfiguration. In *FCCM*, 2016.

[187] Tri M. Nguyen and David Wentzlaff. Morc: A manycore-oriented compressed cache. In *MICRO*, New York, NY, USA, 2015. ACM.

[188] Marco S Nobile, Paolo Cazzaniga, Andrea Tangherloni, and Daniela Besozzi. Graphics processing units in bioinformatics, computational biology and systems biology. *BIB*, 2016.

[189] NVIDIA. NVIDIA Deep Learning Accelerator (NVDLA), 2017.

[190] Jia Pan and Dinesh Manocha. Fast gpu-based locality sensitive hashing for k-nearest neighbor computation. In *GIS*, 2011.

[191] Biswabandan Panda and André Seznec. Dictionary sharing: An efficient cache compression scheme for compressed caches. In *MICRO*, 2016.

[192] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and Joel Emer. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In *ISPASS*, 2019.

[193] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *ISCA*, 2017.

[194] Eunhyeok Park, Sungjoo Yoo, and Peter Vajda. Value-aware quantization for training and inference of neural networks. In *ECCV*, 2018.

[195] Jongsoo Park, Sheng R Li, Wei Wen, Hai Li, Yiran Chen, and Pradeep Dubey. Holistic sparsecnn: Forging the trident of accuracy, speed, and size. *arXiv preprint arXiv:1608.01409*, 2016.

[196] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[197] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 2019.

[198] G. Pekhimenko, T. Huberty, R. Cai, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Exploiting compressed block size as an indicator of future reuse. In *HPCA*, 2015.

[199] Gennady Pekhimenko. *Practical data compression for modern memory hierarchies*. PhD thesis, 2016.

[200] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Michael A. Kozuch, Philip B. Gibbons, and Todd C. Mowry. Base-delta-immediate compression: Practical data compression for on-chip caches. In *PACT*, 2012.

[201] Miguel D. Prado, Maurizio Denna, Luca Benini, and Nuria Pazos. Quenn: Quantization engine for low-power neural networks. *ICCF*, 2018.

[202] Moinuddin K Qureshi, David Thompson, and Yale N Patt. The v-way cache: demand-based associativity via global replacement. 2005.

[203] Alec Radford and Karthik Narasimhan. Improving language understanding by generative pre-training. 2018.

[204] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *ECCV*, 2016.

[205] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. VDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design. In *MICRO*, 2016.

[206] Minsoo Rhu, Mike O'Connor, Niladrish Chatterjee, Jeff Pool, and S. Keckler. Compressing dma engine: Leveraging activation sparsity for training deep neural networks. In *HPCA*, 2018.

[207] Jordan Dotzel Christopher De Sa Zhiru Zhang Ritchie Zhao, Yuwei Hu. Improving neural network quantization without retraining using outlier channel splitting. In *ICML*, 2019.

[208] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *TOS*, 2013.

[209] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *IJCV*, 2015.

[210] R. Russon and Y. Fledel. Ntfs documentation.

[211] Joshua San Miguel, J. Albericio, Natalie Enright Jerger, and A. Jaleel. The Bunker Cache for spatio-value approximation. In *MICRO*, 2016.

[212] Joshua San Miguel, M. Badr, and Natalie Enright Jerger. Load Value Approximation. In *MICRO*, 2014.

[213] Daniel Sanchez and Christos Kozyrakis. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems. In *ISCA*, 2013.

[214] Daniel Sanchez, Luke Yen, Mark D. Hill, and Karthikeyan Sankaralingam. Implementing Signatures for Transactional Memory. In *MICRO*, 2007.

[215] Gurtej S. Sandhu. Emerging memories technology landscape. In *NVMTS*, 2013.

[216] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, 2018.

[217] Guid Sandre, Luca Bettini, Alessandro Pirola, Lionel Marmonier, Marco Pasotti, Massimo Borghi, Paolo Mattavelli, Paola Zuliani, Luca Scotti, Gianfranco Mastracchio, Ferdinando Bedeschi, Roberto Gastaldi, and Roberto Bez. A 90nm 4mb embedded phase-change memory with 1.2v 12ns read access time and 1mb/s write throughput. 2010.

[218] Somayeh Sardashti, Andra Seznec, and David A. Wood. Skewed compressed caches. In *MICRO*, 2014.

[219] Somayeh Sardashti, Andre Seznec, and David A. Wood. Yet another compressed cache: A low-cost yet effective compressed cache. *TACO*, 2016.

[220] Somayeh Sardashti and David A. Wood. Decoupled Compressed Cache: Exploiting Spatial Locality for Energy-optimized Compressed Caching. In *MICRO*, 2013.

[221] Somayeh Sardashti and David A. Wood. Decoupled compressed cache: Exploiting spatial locality for energy optimization. *MICRO (top picks)*, 2014.

[222] Anubhav Savant and Torsten Suel. *Server-Friendly Delta Compression for Efficient Web Access*. Kluwer Academic Publishers, 2004.

[223] Yiannakis Sazeides and James E. Smith. The predictability of data values. In *MICRO*, 1997.

[224] Anil Shanbhag, Nesime Tatbul, David Cohen, and Samuel Madden. Large-scale in-memory analytics on intel® optane™ dc persistent memory. In *DaMon*, 2020.

[225] Sayeh Sharify, Alberto Delmas Lascorz, Mostafa Mahmoud, Milos Nikolic, Kevin Siu, Dylan Malone Stuart, Zissis Poulos, and Andreas Moshovos. Laconic deep learning inference acceleration. In *ISCA*, 2019.

[226] Yongming Shen, Michael Ferdman, and Peter Milder. Overcoming resource underutilization in spatial CNN accelerators. In *FPL*, 2016.

[227] Yongming Shen, Michael Ferdman, and Peter Milder. Maximizing CNN accelerator efficiency through resource partitioning. In *ISCA*, 2017.

[228] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically Characterizing Large Scale Program Behavior. In *ASPLOS*, 2002.

[229] Zekun Ni Xinyu Zhou He Wen Yuheng Zou Shuchang Zhou, Yuxin Wu. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. In *ArXiv*, 2016.

[230] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

[231] Dimitrios Skarlatos, Nam Sung Kim, and Josep Torrellas. Pageforge: A near-memory content-aware page-merging architecture. In *MICRO*, 2017.

[232] Nimit Sharad Sohoni, Christopher Richard Aberger, Megan Leszczynski, Jian Zhang, and Christopher Ré. Low-memory neural network training: A technical report. *arXiv preprint arXiv:1904.10631*, 2019.

[233] Ryan Spring and Anshumali Shrivastava. Scalable and sustainable deep learning via randomized hashing. In *KDD*, 2017.

[234] Nathan Srebro and Tommi Jaakkola. Weighted low-rank approximations. In *ICML*, 2003.

[235] Suraj Srinivas and R. Venkatesh Babu. Data-free parameter pruning for deep neural networks. In *BMC*. British Machine Vision Association, 2015.

[236] James E. Stine, Jun Chen, Ivan Castellanos, Gopal Sundararajan, Mohammad Qayam, Praveen Kumar, Justin Remington, and Sohum Sohoni. FreePDK v2.0: Transitioning VLSI education towards nanometer variation-aware designs. In *MSE*, 2009.

[237] Pengfei Su, Shasha Wen, Hailong Yang, Milind Chabbi, and Xu Liu. Redundant loads: A software inefficiency indicator. In *ICSE*, 2019.

[238] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks. In *FPGA*, 2016.

[239] Torsten Suel. *Delta Compression Techniques*.

[240] Tony Summers. Hardware compression in storage and network attached storage. *SNIA tutorial, Spring*, 2007.

[241] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. Mnasnet: Platform-aware neural architecture search for mobile. In *CVPR*, 2019.

[242] Yingying Tian, Samira M. Khan, Daniel A. Jiménez, and Gabriel H. Loh. Last-level Cache Deduplication. In *ICS*, 2014.

[243] Po-An Tsai and Daniel Sanchez. Compress Objects, Not Cache Lines: An Object-Based Compressed Memory Hierarchy. In *ASPLOS*, 2019.

[244] Yannis Tsividis. *Mixed Analog–Digital VLSI Devices and Technology*. WORLD SCIENTIFIC, 2002.

[245] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. Improving the speed of neural networks on cpus. In *NeurIPS*, 2011.

[246] Jeffrey S. Vetter and Sparsh Mittal. Opportunities for Nonvolatile Memory Systems in Extreme-Scale High-Performance Computing. *CSE*, 2015.

[247] Jeffrey Scott Vitter. Design and analysis of dynamic huffman codes. *J. ACM*, 1987.

[248] Lidong Wang and Cheryl Alexander. Machine learning in big data. *IJMEMS*, 2016.

[249] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *Neurips*, 2016.

[250] Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis. The Case for Compressed Caching in Virtual Memory Systems. In *USENIX ATC*, 1999.

[251] David P. Woodruff. Low rank approximation lower bounds in row-update streams. In *NeurIPS*, 2014.

[252] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C Steely Jr, and Joel Emer. SHiP: Signature-based hit predictor for high performance caching. In *MICRO*, 2011.

[253] Di Wu, Qi Tang, Yongle Zhao, Ming Zhang, Ying Fu, and Debing Zhang. EasyQuant: Post-training Quantization via Scale Optimization. *arXiv preprint arXiv:2006.16669*, 2020.

[254] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. Quantized convolutional neural networks for mobile devices. In *CVPR*, 2016.

[255] Junru Wu, Yue Wang, Zhenyu Wu, Zhangyang Wang, Ashok Veeraraghavan, and Yingyan Lin. Deep k-means: Re-training and parameter sharing with harder cluster assignments for compressing deep convolutions. In *ICML*, 2018.

[256] Yannan Nellie Wu, Joel S. Emer, and Vivienne Sze. Accelergy: An architecture-level energy estimation methodology for accelerator designs. In *ICCAD*, 2019.

[257] Wm A Wulf and Sally A McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH computer architecture news*, 1995.

[258] Wei Pan Xiaofan Lin, Cong Zhao. Towards accurate binary convolutional neural network. In *NeurIPS*, 2017.

[259] Dingqing Yang, Amin Ghasemazar, Xiaowei Ren, Maximilian Golub, Guy Lemieux, and Mieszko Lis. Procrustes: a dataflow and accelerator for sparse deep neural network training. In *MICRO*. IEEE, 2020.

[260] Jun Yang, Youtao Zhang, and R. Gupta. Frequent value compression in data caches. In *MICRO*, 2000.

[261] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *CVPR*, 2017.

[262] Xuan Yang, Mingyu Gao, Jing Pu, Ankita Nayak, Qiaoyi Liu, Steven Emberton Bell, Jeff Ou Setter, Kaidi Cao, Heonjae Ha, Christos Kozyrakis, et al. DNN Dataflow Choice Is Overrated. *arXiv preprint arXiv:1809.04070*, 2018.

[263] Amir Yazdanbakhsh, Divya Mahajan, Hadi Esmaeilzadeh, and Pejman Lotfi-Kamran. AxBench: A Multiplatform Benchmark Suite for Approximate Computing. *IEEE Design Test*, 2017.

[264] Reza Yazdani, Marc Riera, Jose-Maria Arnau, and Antonio González. The Dark Side of DNN Pruning. In *ISCA*, 2018.

[265] Anis Yazidi and Hugo Hammer. Multiplicative update methods for incremental quantile estimation. *IEEE Transactions on Cybernetics*, 2017.

[266] Jianbo Ye, Xin Lu, Zhe Lin, and James Z Wang. Rethinking the smaller-norm-less-informative assumption in channel pruning of convolution layers. *ICLR*, 2018.

[267] Haoran You, Xiaohan Chen, Yongan Zhang, Chaojian Li, Sicheng Li, Zihao Liu, Zhangyang Wang, and Yingyan Lin. Shiftaddnet: A hardware-inspired deep network, 2020.

[268] Vinson Young, Sanjay Kariyappa, and Moinuddin K. Qureshi. Enabling transparent memory-compression for commodity memory systems. In *HPCA*, 2019.

[269] Vinson Young, Prashant J. Nair, and Moinuddin K. Qureshi. DICE: Compressing DRAM Caches for Bandwidth and Capacity. In *ISCA*, 2017.

[270] Shimeng Yu and Pai-Yu Chen. Emerging memory technologies: Recent trends and prospects. *ISSCM*, 2016.

[271] Xiyu Yu, Tongliang Liu, Xinchao Wang, and Dacheng Tao. On compressing deep models by low rank and sparse decomposition. In *CVPR*, 2017.

[272] Sergey Zagoruyko. Torch | 92.45% on CIFAR-10 in Torch, 2015.

[273] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. In *BMVC*, 2016.

[274] Shuangfei Zhai, Yu Cheng, Weining Lu, and Zhongfei Zhang. Doubly convolutional neural networks. In *NeurIPS*, 2016.

[275] Can Zhang, Yaming Xu, Hongliang Wang, Wei Liu, Qi Mu, and Wei Guo. An offloading architecture of lossless compression based on smart nic. In *ICCEIC*, 2020.

[276] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *FPGA*, 2015.

[277] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 2015.

[278] Jiaqi Zhang, Xiangru Chen, Mingcong Song, and Tao Li. Eager pruning: algorithm and architecture support for fast training of deep neural networks. In *ISCA*, 2019.

[279] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-X: An accelerator for sparse neural networks. In *MICRO*, 2016.

[280] Tianyun Zhang, Shaokai Ye, Yipeng Zhang, Yanzhi Wang, and Makan Fardad. Systematic Weight Pruning of DNNs using Alternating Direction Method of Multipliers. In *ICLR*, 2018.

[281] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *CVPR*, 2018.

[282] Xuebin Zhang, Jiangpeng Li, Hao Wang, Danni Xiong, Jerry Qu, Hyunsuk Shin, Jung Pill Kim, and Tong Zhang. Realizing transparent os/apps compression in mobile devices at zero latency overhead. *TC*, 2017.

[283] Jie Zheng and Jun Luo. A PG-LSH Similarity Search Method for Cloud Storage. In *CIS*, 2013.

[284] Hao Zhou, Jose M. Alvarez, editor="Leibe Bastian Porikli, Fatih", Jiri Matas, Nicu Sebe, and Max Welling. Less Is More: Towards Compact CNNs. In *ECCV*, 2016.

[285] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 1977.

[286] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 1978.

[287] Neta Zmora, Guy Jacob, Lev Zlotnik, Bar Elharar, and Gal Novik. Neural network distiller: A python package for DNN compression research. *arXiv preprint arXiv:1910.12232*, 2019.

[288] Aviad Zuck, Sivan Toledo, Dmitry Sotnikov, and Danny Harnik. Compression and SSDs: Where and how? In *INFLOW*, 2014.