A Framework for Managing Heterogeneous Memory for Large Scale Machine Learning Workloads

By

MARK HILDEBRAND
DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Venkatesh Akella, Chair

Jason Lowe-Power

Soheil Ghiasi

Committee in Charge

2022

Contents

Abstract iv
Acknowledgments vi

Chapter 1. Introduction 1
1.1. Introduction 1
1.2. Intel Optane DC PMM 2
1.3. Research Motivation 5
1.4. Dissertation Contributions and Organization 6

Chapter 2. Limitations of Hardware Managed Gigascale DRAM Caches 8
2.1. Introduction 8
2.2. Background and Related Work 10
2.3. Evaluation Methodology and Validation 10
2.4. DRAM Cache / 2LM Mode 15
2.5. Case Study 1: Convolutional Neural Networks 20
2.6. Case Study 2: Graph Processing 25
2.7. Discussion and Mitigation Strategies 29

Chapter 3. Compiler-Based Heterogeneous Memory Management for Statically Analyzable Workloads 31
3.1. Introduction 31
3.2. Background 33
3.3. AutoTM 36
3.4. Implementation Details 41
3.5. Evaluation Methodology 43
3.6. Results 45
3.7. Extending AutoTM 53
3.8. Related Work 57
3.9. Conclusions 58

Chapter 4. Generalizing Heterogeneous Memory Management 60
4.1. Introduction 60
4.2. A Generic Heterogeneous Memory Management System 61
4.3. Basis for the Data Manager and Modular Policy 69
4.4. *CachedArrays* Implementation 77
4.5. Annotations for CNN Workloads 81
4.6. Evaluation Methodology 85
4.7. Results 90
4.8. Related Work 95

4.9. Discussion 96

Chapter 5. DLRM Case Study 100
5.1. Introduction 100
5.2. Embedding Table Implementation 102
5.3. Embedding Table Experiments 107
5.4. Software Caches for Gigascale Embedding Tables 116
5.5. DLRM Implementation 123
5.6. End-to-End DLRM Performance of *CachedEmbeddings* 126
5.7. Related Work 130

Chapter 6. Conclusions and Future Work 132
6.1. Limitations 132
6.2. Hardware Support for Data Tiering 133

Appendix A. OneDNN Wrapper 136
A.1. OneDNN Summary 136
A.2. Exposing the C API 137
A.3. Exposing Types to Julia 140
A.4. Putting it all Together 142

Bibliography 144

Abstract

The memory requirements of emerging applications, especially in the domain of machine learning workloads, is outpacing the capacity of traditional memory devices like DRAM. At the same time, heterogeneity in the memory hierarchy is emerging on multiple fronts both with high-capacity, low-bandwidth devices like Intel Optane Data-Center (DC) Persistent Memory Modules (PMM), and low-capacity, high-bandwidth devices like High Bandwidth Memory (HBM). A fundamental question introduced by this heterogeneity is: how do we efficiently manage application data to fully exploit the properties of the underlying memory technologies? This work explores techniques and ideas towards answering this question and understanding the performance implications of heterogeneous memory.

First, Intel's DRAM cache mode for Optane DC is reverse engineered using a suite of micro-benchmarks and large scale machine learning applications. It is discovered that for machine learning training applications with large memory footprints and large-scale graph analytics, the DRAM cache behaves poorly with significant access amplification and low bandwidth utilization. There are three reasons for this performance degradation: (1) inflexible direct mapped policy leading to conflict misses, (2) poor traffic shaping cause by on-demand accesses and metadata management, and (3) lack of program semantic insight leading to many unnecessary and slow dirty data writebacks.

Next, AutoTM, a profile-guided compiler-based optimization technique that uses Integer Linear Programming to derive optimal tensor placement and movement for machine learning training in heterogeneous memory systems, is presented. The *nGraph* compiler is modified to implement AutoTM for two different systems: a CPU-based system with a combination of DRAM and Optane DC and a GPU-based system capable of using both GPU and CPU memory For DRAM/Optane DC systems, AutoTM outperforms the DRAM cache by as much as 3× and as much as 4× for the transparent *cudaMallocManaged* for GPU/CPU systems.

The third part of this work generalizes memory management primitives. A generic heterogeneous memory management framework can be broken into three parts: the *system* (the entity responsible for managing data and metadata), the *policy* (the entity orchestrating the placement and movement of data), and the *abstract runtime* (the application or runtime that is actually using the data). The key insight is the modularity of this organization. Upon this framework is built

CachedArrays, a *policy/system* package implemented in the Julia programming language. Unlike AutoTM, *CachedArrays* works for applications with dynamic control flow and improves end-to-end convolutional neural network (CNN) training performance by up to 2× over the DRAM cache.

Finally, to demonstrate the generality of this framework, it is applied to gigabyte scale embedding tables for large DLRM workloads. A performance analysis of the design space embedding table lookup and update operations on Xeon CPUs is conducted. This leads to the implementation of *CachedEmbeddings*, an instance of the generic heterogeneous memory management framework optimized for small-sized random memory accesses. Using a high-performance DLRM implementation, *CachedEmbeddings* out performs the DRAM cache for end-to-end DLRM training by up to 1.45× by using a modular, distribution-dependant policy.

Acknowledgments

This dissertation would never have seen the light of day were it not the selfless aid of many individuals along the way.

First, I wish to express my sincere gratitude towards my advisors and mentors Professor Venkatesh Akella and Professor Jason Lowe-Power. Venkatesh recruited my to work with something called “Optane” at a crucial point in my degree adventure, and I’ll forever be grateful for his patience, guidance, and expertise, both for this dissertation and for the papers published along the way. Jason is the one of the best researchers I have ever met and is a role model professor in all areas. I am very grateful to have worked with him.

Special thanks to Professor Soheil Ghiasi for serving on my dissertation and qualifying examination committees. I appreciate your valuable feedback and insightful questions. I also wish to thank Professors Zhou Yu and Houman Homayoun for serving on my qualifying examination committee.

This work would not have been possible without the support Jawad B. Khan and Sanjeev Trika, the project’s sponsors at Intel. I deeply appreciate their expertise, feedback, and insights regarding Optane DC memory. It was a joy working with you two! I would also like to thank the Intel corporation and especially the IT professionals working there for providing access to Optane equipped servers for running experiments.

Many thanks to Terry O’Neill for his mentorship and worldly wisdom. I am also thankful for Julian T. Angeles for our collaboration in understanding the implications of Intel’s DRAM cache and for our adventures in general programming mayhem. Thanks should also go to Bobby Bruce for his endless encouragement and bottomless pot of coffee.

Thanks should also go to the other members of the Davis Architecture Research Group not previously mentioned, including but by no means limited to Mahyar Samani, Maryam Babaie, Ayaz Akram, Hoa Nguyen, Kaustav Goswami, Toluwanimi Odemuyiwa, Marjan Fariborz, Kramer Straube, Professor Matthew Farrens, Professor Chris Nitta, Melissa Katherine Jost, Kelly Nguyen, Bradley Wang, Nikitha Muddireddy, and Nima Ganjehloo.

I would also like to acknowledge the members of the VLSI Computation Lab where I spent the first few years of my PhD journey. In particular, Professor Bevan Baas, Brent Bohnenstiehl, Timothy Andreas, Satyabrata Sarangi, Shifu Wu, Professor Aaron Stillmaker, Jon Pimentel, Arthur Hlaing, and Filipe Borges.

I want to express my deepest gratitude towards my friend Matthew Toney. I do not think I would have seen this through without your support and steadfast example. You are an inspiration Matt!

I also want to thank Fishstix for all the joy the furry feline brought into my life. May she rest in peace.

Thank you to all my coworkers at the Rocknasium for creating a comfortable and supportive atmosphere. It has been such a blessing to be able to retreat to the climbing gym over these years.

Finally, I wish to extend my sincerest appreciation towards my brother and parents for all their support. To my father, Brent Hildebrand, in particular, who passed away before I finished my degree: thank you for everything. I will try to live my life with honor and integrity, as you did yours.

CHAPTER 1

Introduction

1.1. Introduction

For years, researchers, scientists, and engineers have been claiming the death of Moore's Law, the observation of exponential growth in silicon-based transistor density. And while foundries like TSMC continue to extract more performance out of silicon, it is undeniable that we have entered a new age of computer architecture design and programming paradigms. Programmers can not rely on general purpose hardware simply getting faster to fulfill the needs of modern data-center and machine learning workloads. Instead, we've entered an era of specialization.

In particular, the memory requirement of modern applications, particularly those related to machine learning (ML), are outpacing the improvement in capacities of memory devices. Often brought up in discussions about increasing memory pressure are the absurdly large language models being developed by the likes of OpenAI, Microsoft, and NVidia. For instance, in early February 2020, Microsoft announced Turing-NLG[1], a 17-billion parameter language model. Later that same year, OpenAI announced the 175-billion parameter GPT-3 in 2020 [**10**]. Not to be out-done, Microsoft teamed up with NVidia to produce Megatron-Turing NLG in late 2021, with a memory footprint of nearly 17 TB required for training [**98**]. It can be debated whether further language model inflation is the best way of improving model performance versus, for example, improving model architecture. Nevertheless, this provides an example of workloads taxing the memory capacity of server-scale computers.

Deploying and training such models requires significant infrastructure, consisting of hundreds of GPUs. To illustrate how extreme this gets, results from MLPerf Training 0.7 [**63**] (a benchmark suite for comparing system performance for Machine Learning (ML) models) shows Nvidia distributing training to over 2000 GPUs and Google using over 4000 TPU-v3[2]. The cost of these large

[1]https://www.microsoft.com/en-us/research/blog/turing-nlg-a-17-billion-parameter-language-model-by-microsoft/
[2]`https://mlperf.org/training-results-0-7`

scale deployments is such that only the largest companies are capable of pushing the boundaries in large scale ML research.

As models continue to grow, memory capacity and bandwidth will continue to be key bottlenecks. Existing approaches to distribute these models such as *data parallelism* and *model parallelism* might not be sufficient. For example, data parallelism requires large batchsizes to achieve high computational performance on each worker. However, too large a batchsize may require *more* time to train a network due to poor convergence [**64**]. Model parallelism is promising, but brings with it its own challenges including effectively managing communication delay and synchronization.

At the same time, the computing industry is moving more and more towards heterogeneity in the memory system [**8**, **39**, **48**, **101**]. This heterogeneity introduces different memory technologies within the same computing system, with various trade-offs. One such example is Intel Optane Data-Center (DC) [**39**], an emerging non-volatile memory technology offering significantly higher memory capacities than DRAM albeit with lower performance. Applications with increasing memory requirement may take advantage of the capacity offered by this technology, but only if doing so does not incur an unacceptable performance penalty. In this dissertation, we will investigate techniques for managing heterogeneous memory systems, for deciding when to place data in a small fast memory (e.g., DRAM) or a large slow memory (e.g., Optane DC) to utilize the advantages of both. Thanks to the generosity of this project's sponsors, a real DRAM/Optane DC based system will be used as a test-bed for the ideas presented within this dissertation. However, it is hoped that these ideas generalize to the broader landscape of emerging heterogeneous memory systems.

1.2. Intel Optane DC PMM

Intel Optane Data-Center (DC) Persistent Memory Modules (PMM) are non-volatile random access memory (NVRAM) devices based on phase change memory designed for direct load/store random access similar to DRAM [**39**, **48**]. The first two generations of these devices were designed to live on the DRAM bus, using a modified protocol called "DDRT". Due to the modified DDRT protocol, Optane DC required CPUs with modified memory controllers. Because of their residence on the DDR bus, direct loads and stores to these devices are possible.

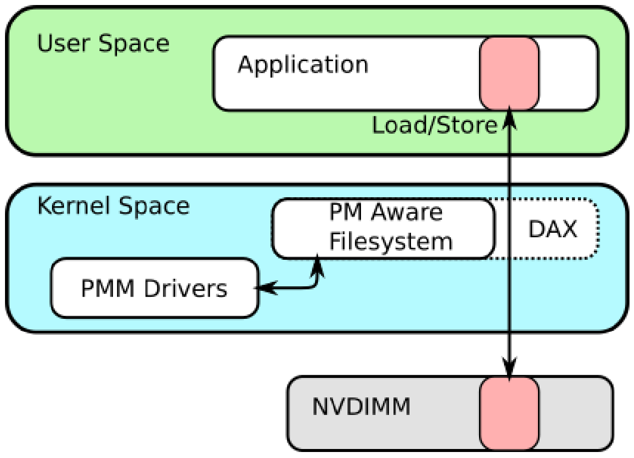


FIGURE 1.1. App Direct persistent memory programming model.

Low-level configuration provisioning of these devices is performed through the **`ipmctl`**[3] tool and the **`ndctl`**[4] tool is used to expose these devices to the host operating system and user-level applications. There are three main ways in which persistent memory (PM) is available to applications.

1.2.1. App Direct. Unlike traditional block-based storage devices, persistent memory (PM) devices support "direct access" (DAX) file systems (Figure 1.1). Using this type of file system changes the behavior of memory mapping files. Instead of maintaining a DRAM buffer, the kernel instead directly memory maps the backing file into the user application. After mapping, any loads and stores to the corresponding virtual addresses go directly to the underlying media, bypassing kernel layers entirely [**48**].

When used as *volatile* memory (i.e., persistence is not required), an application can treat the memory obtained from this memory mapping exactly as it would normal memory. However, if persistence *is* required, than care needs to be taken by the application to preserve data integrity in the case of an interrupting event. For example, a power failure midway through a large write can result in a torn write, leaving the file on persistent memory in an inconsistent state. To that end, libraries like PMDK[5] and new x86 hardware instructions like **`clflushopt`** and **`clwb`** (described in Table 1.1) allow write to PM to occur transactionally (i.e., either completely succeed or completely fail).

[3]**`https://github.com/intel/ipmctl`**
[4]**`https://github.com/pmem/ndctl`**
[5]**`https://github.com/pmem/pmdk`**

Instruction	Description
`clflushopt`	Evict the corresponding cache line from all levels of the CPU cache hierarchy. Unlike the older **`clflush`** operation, **`clflushopt`** may be executed out of order with other **`clflushopt`** instructions, leading to potentially better performance of many such instructions.
`clwb`	Write back the corresponding cache line to main memory. The cache line may be retained in the CPU cache hierarchy.

Table 1.1. Extensions to the x86 ISA to support persistent memory. When cache line writes destined for PM reach the memory controller, they enter the persistency domain where they are guaranteed to commit even in the event of a power failure [**81**].

The focus of this dissertation is on using PM as volatile memory for bandwidth intense applications. As such, we will not explore the implications of PM transactions. While conceptually simple to use in an application, Optane DC cannot serve as a drop-in replacement for DRAM as load/store latency is on the order of 3× higher than DRAM with 60% lower bandwidth [**109**]. What these devices offer, instead is capacity with up to 512 GB per module. At this size, a Xeon CPU with 6 memory channels can have up to 3 TB of random access persistent memory in addition to DRAM.

Because of these trade-offs (higher capacity, but lower bandwidth/higher latency), Optane DC serve as an excellent real-world test case for heterogeneous data tiering and management.

1.2.2. Memory Mode (2LM). Persistent memory modules can be used in the so-called 2LM (also known as *memory mode* or *cached*) [**48**], where PM act transparently as system memory. In this mode, system DRAM serves as a direct mapped cache for the non-volatile memory. The access granularity of this cache is 64B, matching the cache line size of the underlying CPU. While not mentioned explicitly, Intel patents suggest that cache tags are stored along with ECC data [**84**]. ECC DRAM is implemented by adding an extra DRAM module to each DIMM. Thus, each 64B data transaction for each DIMM is accompanied by 8 B (64 bits) of ECC. Of these 64 bits of ECC data, only 20 [**13**] are required to provide Single Error Correction/Double Error Detection redundancy, leaving ample room for tag metadata, including both physical address and cache line state. Our data is consistent with this approach.

This operating mode was likely introduced as a way to lower the barrier to adoption of PM, though only for the capacity aspect and not for persistence. For example, when operating in 2LM, a system like that described above may see as much as 3 TB of memory per socket. DRAM caches have been well studied in simulation [**17**,**18**,**50**,**51**,**59**,**61**,**80**]. However, these previous works have not taken all of the realistic implementation details (e.g., tracking "coherence" of request issued to PM) leaving gaps between research proposals and the actual implementation. Chapter 2 will discuss the design of this DRAM cache in detail and outline many of the performance problems introduced by such a cache.

1.2.3. PM as a NUMA Node. A third way to use to PM as volatile memory is to use the `daxctl`[6] tool expose the persistent memory modules as extra NUMA nodes. This allows existing NUMA aware allocation and memory pinning to take advantage of the extra capacity offered by PM. However, this still does not address the main question: how to efficiently use this memory?

1.3. Research Motivation

The question to answer in heterogeneous memory systems, regardless of whether it's for machine learning or more general applications, is where to place data and when should it be moved between memory devices. One approach is to simply use something like the transparent DRAM cache described in the previous section. Unfortunately, as we will show in Chapter 2, this can lead to sub-optimal performance. For machine learning training on GPUs, bespoke solutions like vDNN [**89**] and ZeRO [**83**] use application specific knowledge to answer this question. However, these specialized approaches approaches only apply to heterogeneous memory systems where only memory in the "device local" memory location is directly byte-level accessible. This is quite different from what emerging heterogeneous memory systems will look like.

1.3.1. Examples of Emerging Heterogeneous Memory Systems. Recent advances in interconnect technology and device packaging have lead to new classes of heterogeneous memory systems. One example is the previously mentioned Optane DC, which allow byte-level access through the DDR bus. Additionally, emerging technologies like CXL will extend byte-level addressability and cache-coherence to fabric-attached devices [**101**]. New generations of Intel CPUs like Sapphire Rapids, in addition to CXL attach Optane PM, will also include on-package high-bandwidth memory (HBM) for yet another level of heterogeneous memory. [**8**].

[6]`https://github.com/pmem/ndctl/tree/main/daxctl`

All of these emerging technologies and many more like them will involve different trade-offs and decisions to be made when it comes to heterogeneous memory management. Existing solutions like block-based caches [**43**, **48**], virtual page-management [**54**, **108**], and manual programmer development [**33**] are either not efficient enough or scalable enough. Furthermore, while tools like Intel's Persistent Memory Developement Kit[7] and Samsung's recent Scalable Memory Development Kit[8] are providing tools for exposing heterogeneous memory to an application, there's still the question of how to use this memory effectively.

1.3.2. Summary. In this work, we seek to understand some of the potential performance pitfalls associated with Optane PM and develop techniques for scalably managing the proliferation of emerging heterogeneous memory technologies. While the bulk of this dissertation targets a heterogeneous CPU-based system with DRAM and Optane PM targeting machine learning workloads, the goal is for the techniques and insights developed here to generalize to other similar systems as well. For example, in Chapter 3, we will show that the techniques developed indeed generalize to a GPU-CPU heterogeneous system. Finally, while extracting the best performance of a system requires a detailed understanding of the memory technologies involved and their respective trade-offs, we will show in Chapters 4 and 5.1 that a common approach to developing a heterogeneous memory management framework can be used for two very different classes of problems.

1.4. Dissertation Contributions and Organization

The rest of this dissertation is organized as follows. Chapter 2 reverse engineers the so called "2LM" DRAM cache implementation on Intel Cascade-Lake servers. Using several machine learning and graph processing benchmarks running on real hardware, it is shown that the hardware only nature of this approach to heterogeneous memory management suffers in a number of key areas such as low bandwidth utilization and a lack of program semantic information.

Chapter 3 introduces AutoTM, a Integer Linear Programming (ILP) based technique for optimizing memory location and movement between DRAM and Optane PM for CNN training. This technique yields up to a 3x performance over the native 2LM hardware DRAM cache. AutoTM is further extended to manage GPU-CPU memory for GPU-based training.

[7]https://github.com/pmem/pmdk
[8]https://github.com/OpenMPDK/SMDK

Chapter 4, we demonstrate a generalized runtime framework for managing heterogeneous memory. A generalized API for a multi-level memory manager is presented and implemented in the Julia programming language. Using this approach, the dependency for incorporating heterogeneous memory into application development can be inverted. That is, this framework may be used as a compositional building block for application development rather than developing bespoke memory management solutions for applications, such as the case in AutoTM.

Chapter 5.1 builds on the memory management API to implement two-level memory management for the fine-grained memory access pattern found Deep Learning Recommendation Model (DLRM) embedding table lookups. An abstract API for embedding table lookup and updates is presented the DLRM implemented with it outperforms a state-of-the-art PyTorch implementation from Intel.

CHAPTER 2

Limitations of Hardware Managed Gigascale DRAM Caches

2.1. Introduction

Large scale machine learning and large scale graph analytics represent workloads of interest for high performance server in the forseeable future. Emerging machine learning models in NLP and recommendation engines (such as GPT3 [**10**] and DLRM [**67**]) can have over 100 billion parameters requiring hundreds of gigabytes to terabytes of memory for training. Similarly real world graphs can have hundreds of billions of edges, requiring hundreds of gigabytes to just store the graphs [**72**]. As a result, the cost of memory (DRAM) is becoming an important concern in datacenters and other high performance computing facilities dealing with large scale data analysis [**30**, **31**].

To address this challenge Intel, as mentioned in Chapter 1, introduced Optane Data-Center Persistent-Memory-Modules (DC PMM), a persistent memory (PM)[1]) technology based on phase change memory that can serve as a drop-in replacement for conventional DRAM [**39**]. While programmers can use the PM as a main memory DRAM replacement using normal load and store instructions, the latency is 3× higher and the bandwidth is at least 60% lower than DRAM [**109**]. Traditionally, to hide high memory latency and limited bandwidth, computer architects have turned to hardware caches. In this tradition, Intel Cascade Lake systems implement a *DRAM* cache for PM. DRAM caches have been well studied in simulation [**17**, **18**, **50**, **51**, **59**, **61**, **80**]. These previous works have not taken all of the realistic implementation details (e.g., tracking "coherence" of request issued to PM) leaving gaps between research proposals and the actual implementation.

In this chapter, we analyze the performance of an *actual implementation* of the DRAM cache in Intel's Cascade Lake based servers on workloads whose memory footprint *greatly exceeds* the capacity of DRAM. We first analyze the behavior of the DRAM cache with microbenchmarks to reverse engineer its design and understand pathological performance cliffs. It is well known that

[1]This chapter uses the term "PM" to refer to memory located on the Optane DC DIMMs.

this DRAM cache is implemented as a direct-mapped [**48**], and we find that the tags are stored ECC bits of the DRAM DIMMs to limits the access overhead. However, we also find that in many cases there are extra DRAM accesses required to update the cache metadata (e.g., tag reads before writes) which can significantly decrease the performance of miss-heavy workloads. In fact, using microbenchmarks on real hardware, we find that a single demand request can require up to **5 memory accesses**.

After using microbenchmarks to understand the cache behavior and implementation, we analyze two memory capacity limited workloads: training large convolutional neural networks (CNNs) [**41**, **46**, **97**, **100**] and graph analytics [**33**]. We show that in these realistic workloads, the DRAM cache *can hurt performance* even with a modest cache miss rate. We show that for the CNN workload, software management can increase performance by up to 3× over the DRAM cache. Furthermore, we show significant access amplification and bandwidth reduction for graph based workloads.

Fundamentally, we find three characteristics of this DRAM cache implementation which causes performance degradation for workloads with large working sets.

(1) The direct-mapped, insert on miss cache is inflexible and many conflicts can increase the miss rate.
(2) Under high miss rates, memory bandwidth is poorly utilized with extra bandwidth used for non-demand accesses (e.g., fills, writebacks, and tag checks).
(3) For some workloads the data in the DRAM cache is *temporary* or *dead* from the program's perspective leading to wasted data movement.

While some of these characteristics may be alleviated in future hardware, we can use these three insights *on today's hardware* to improve the performance of heterogeneous memory systems.

The rest of the chapter is organized as follows. We start with the quick background on Intel's PM technology and related work in the area of benchmarking PM from recent literature. In Section 2.3 we present the details of the evaluation and validation framework. Section 2.4 follows up with a detailed analysis of the DRAM cache in these systems. Next we use two representative case studies from deep learning and graph analytics to corroborate the findings from the microbenchmark experiments. We end the chapter with a discussion of the results and ideas for software based mitigation strategies in Section 2.7.

2.2. Background and Related Work

There have been several efforts in research literature that focus on evaluating the system level performance of Optane DC [**48**, **76**, **77**, **92**, **102**], especially in comparison with DRAM. More recently, Wang et. al [**107**] developed a profiler and PM simulator to model the microarchitecture of PM in general. However, to the best of our knowledge there has been no effort in trying *understand* the performance of DRAM caches in large scale PM-based systems. However, the tools described by Wang [**107**] *could* be used for hardware/software codesign of DRAM caches in the future, building on the findings in this chapter.

On the application front there has been work on the design of data structures and algorithms to mitigate the disadvantages of PM, chiefly the slower and asymmetric read/write latency and bandwidth [**12**, **26**, **71**, **75**, **95**]. Dhulipala et. al [**26**] and Gill et. al [**33**] evaluate the performance of large scale graph analytics on PM based systems. These works focus on application performance evaluation and optimization but do not delve into the details of behavior of the DRAM cache (the 2LM mode) and why they do not work well on these applications. The goal of this work is to fill this gap. In fact, one could view Sage [**26**] as a software technique to mitigate the limitations of DRAM caches in PM based systems as discussed in Section 2.6 and Section 2.7

2.3. Evaluation Methodology and Validation

2.3.1. Test System. Our test machine is a two-socket Xeon server (illustrated in Figure 2.1) equipped with 24-core Cascade Lake engineering sample CPUs. The CPU on each socket has two integrated memory controllers (IMC), each with three memory channels. Integrated memory controllers are responsible for performing the actual reads and writes to DRAM and PM. Each memory channel is populated with a 32 GiB DDR4 DRAM DIMM and a 512 GiB Optane DC DIMM.

2.3.2. Evaluation Methodology.

2.3.2.1. *Kernel Benchmark Generator.* To test the basic bandwidth performance of DRAM and PM, both in 1LM and 2LM, we made a custom open source benchmark generator[2] written in Julia [**7**]. The generator uses Julia's metaprogramming and just-in-time compilation to generate

[2]`https://github.com/darchr/KernelBenchmarks.jl`

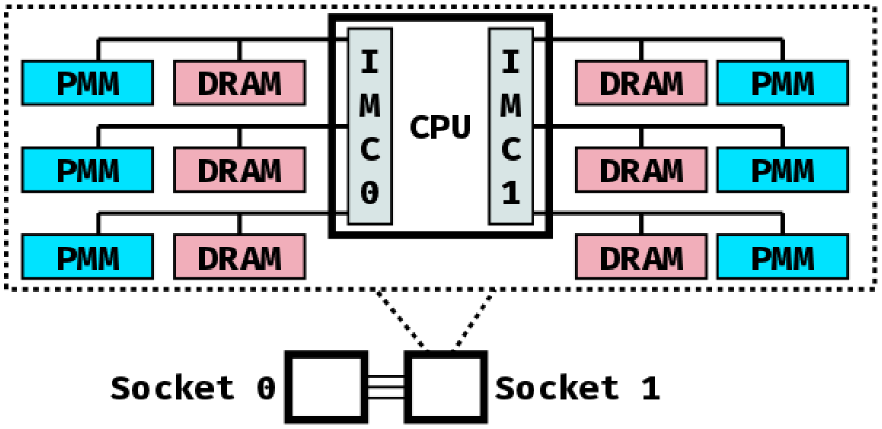


FIGURE 2.1. Diagram of our test platform. Each socket has 192 GiB of DRAM and 3 TB of PM spread across six memory channels.

custom low overhead load and store loops. Memory can be accessed either sequentially or pseudo-randomly. When accessed pseudo-randomly, we ensure that each addresses is touched exactly once (i.e. no repeats) using a maximum length Linear Feedback Shift Register to generate array indices. Furthermore, for pseudo-random iteration, access granularity ranges from 64 B to 512 B. We found sequential iteration is largely indifferent to access granularity, so only a single result for sequential access is reported. For these experiments, we used read-only, write-only, and read-modify-write operations. We explore both *standard* or *nontemporal* instructions for all stores. *Nontemporal* stores bypass the on-chip cache, allowing us to directly study the behavior of LLC writes to the memory controller. Data is partitioned evenly across threads when multithreading is used.

Listing 2.1 shows the generated x86 code for a 512 B, nontemporal write-only workload that pseudo-randomly accesses array addresses using a LFSR. Lines 1-3 load arguments where `r8` is the base pointer for the array segment and `r9` is the starting seed for the LFSR. The write-only data is broadcast to the 64 B AVX512 register (`zmm0`) in lines 4 and 5. The main loop body encompases lines 7 to 32. The 512 B write takes 8 nontemporal vector stores (`vmovnts`). Lines 22 to 32 implement the LFSR as annotated in the listing. Listing 2.2 illustrates x86 assembly for a sequential read-modify-write kernel that uses 16 B loads and stores. As can be seen, the generated assembly for these microbenchmarks is minimal, providing high-performance implementations of the kernels in question.

```
    mov rax, qword ptr [rdi]
    mov r8, qword ptr [rax]
    mov r9, qword ptr [rsi + 16]
    movabs  rax, offset .rodata.cst4
    vbroadcastss    zmm0, dword ptr [rax]
    mov rdx, r9
L29:
    mov rax, rdx
    shl rax, 9
    vmovntps    zmmword ptr [rax + r8 - 512], zmm0 # + 512 B contiguous write.
    vmovntps    zmmword ptr [rax + r8 - 448], zmm0 # |
    vmovntps    zmmword ptr [rax + r8 - 384], zmm0 # |
    vmovntps    zmmword ptr [rax + r8 - 320], zmm0 # |
    vmovntps    zmmword ptr [rax + r8 - 256], zmm0 # |
    vmovntps    zmmword ptr [rax + r8 - 192], zmm0 # |
    vmovntps    zmmword ptr [rax + r8 - 128], zmm0 # |
    vmovntps    zmmword ptr [rax + r8 - 64], zmm0  # +
    mov rdi, qword ptr [rsi]
    mov rax, qword ptr [rsi + 8]
    nop dword ptr [rax + rax]
L112:
    mov ecx, edx # + Generate the next term in the LFSR sequence.
    and ecx, 1   # |
    neg rcx      # |
    and rcx, rax # |
    sar rdx      # |
    xor rdx, rcx # +
    cmp rdx, r9  # + Compare with seed (if equal, done with iteration).
    je  L141
    cmp rdx, rdi # + Compare with max length. If exceeding, iterate again to yield an
    jg  L112     # + in-bounds index.
    jmp L29
L141:
    vzeroupper
    ret
    nop word ptr cs:[rax + rax]
```

Listing 2.1. Auto Generated code for 512B non-temporal writes using pseudo-random array indexing.

2.3.2.2. *Hardware Performance Counters.* To measure DRAM and PM traffic, we use uncore hardware performance counters located in each IMC. These counters capture column access strobes (CAS) for DRAM reads and writes. The Cascade Lake generation added IMC counters for PM read and write requests, and 2LM tag statistics including tag hit, tag miss clean, and tag miss dirty, which will be explained in more detail later. Event codes and masks for the counters used in this work are given in Table 2.1. Results from the hardware performance counters are validated with the expected data movement and benchmark wall clock time.

```
    mov rcx, qword ptr [rsi]
    mov rax, qword ptr [rsi + 8]
    sub rax, rcx
    jl  L81
    mov rdx, qword ptr [rdi]     # + Bounds check and prepare loop induction variable.
    mov rdx, qword ptr [rdx]     # |
    mov rsi, qword ptr [rdi + 8] # |
    shl rsi, 2                   # |
    shl rcx, 4                   # |
    add rcx, rsi                 # |
    add rcx, rdx                 # |
    add rcx, -20                 # |
    inc rax                      # +
    movabs  rdx, offset .rodata.cst4      # + Broadcast increment variable to register.
    vbroadcastss    xmm0, dword ptr [rdx] # +
    nop word ptr [rax + rax]
L64:
    vaddps  xmm1, xmm0, xmmword ptr [rcx] # + Main loop body.
    vmovaps xmmword ptr [rcx], xmm1       # |
    add rcx, 16                           # |
    dec rax                               # |
    jne L64                               # +
L81:
    ret
    nop word ptr cs:[rax + rax]
```

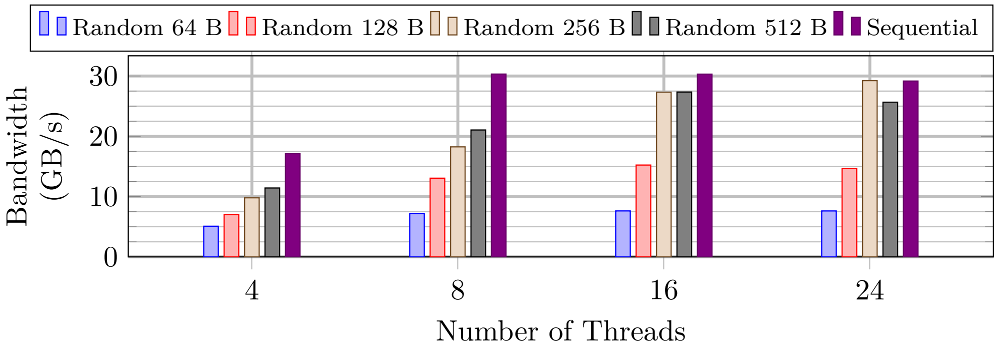
LISTING 2.2. Generated code for a sequential read-modify-write kernel using 16 B loads and stores.

Metric	Event Code	Umask
DRAM Reads	0x04	0x3
DRAM Writes	0x04	0xC
PM Reads	0xEA	0x2
PM Writes	0xEA	0x4
2LM Tag Hit	0xD3	0x1
2LM Tag Miss Clean	0xD3	0x2
2LM Tag Miss Dirty	0xD3	0x4

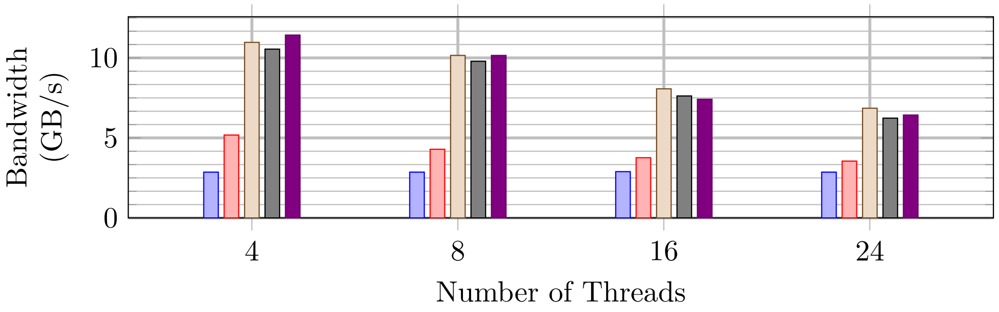
TABLE 2.1. Event codes and umasks for IMC performance counters used to gather data on DRAM, PM, and 2LM behavior. The events listed here all count 64B transactions. Events highlighed in bold were introduced in the Cascade Lake generation of processors.

Each benchmark was executed on a quiet system. Unless otherwise specified, all six Optane DC DIMMs are configured as a single interleaved set and experiments are run on a single socket to avoid NUMA overheads.

2.3.3. PM Performance Results. The results obtained here are in line with observations made by other researchers [**33**, **48**, **76**, **92**]. We highlight results that are relevant to our upcoming



(a) Read bandwidth using *standard* load instructions.



(b) Write bandwidth using *nontemporal* store instructions.

FIGURE 2.2. Bandwidth to 6 interleaved 512 GiB PM DIMMs.

discussion in Section 2.4 on the 2LM DRAM cache. Since read and write bandwidth to Optane DC is asymmetric, we will consider these separately. Figure 2.2a shows the read bandwidth of six interleaved 512 GB PM modules under varying thread counts. Sequential bandwidth scales with the number of threads up to a maximum 30 GB/s with 8 threads, at which it stops increasing. This result is slightly different than the 39 GB/s reported in other works [**48**] because our system uses 512 GiB DIMMs instead of 128 GiB or 256 GiB DIMMs. The 512 GiB DIMMs provide a maximum read bandwidth of 5.3 GB/s read bandwidth per DIMM while the others provide 6.8 GB/s [**21**].

Figure 2.2b demonstrates the write bandwidth of PM when using *nontemporal* stores. In addition to bypassing the on-chip cache, *nontemporal* stores do not need a Read-For-Ownership (RFO), a step in Intel's usual cache coherence protocol [**22**], and are critical for high PM write bandwidth [**109**]. Write bandwidth peaks with four threads, and is roughly the same for sequential and random access exceeding 256 B. Limited buffer space within the Optane DIMM decreases the media controller's ability to merge sequential 64 B writes into a single 256 B write, leading to write

Table 2.2. Summary of generated reads and writes for 2LM. Figure 2.3 shows details of why these requests generate these actions. The dirty data optimization (DDO) allows the IMC to elide the tag check for some writes.

	LLC Read			LLC Write			
	Hit	Miss		Hit	Miss		DDO
		Clean	Dirty		Clean	Dirty	
DRAM Read	1	1	1	1	1	1	
DRAM Write		1	1	1	2	2	1
PM Read		1	1		1	1	
PM Write			1			1	
Amplification	**1**	**3**	**4**	**2**	**4**	**5**	**1**

amplification and the observed drop in bandwidth [**109**]. In summary, with this system we can achieve just over to 30 GB/s read and 11 GB/s write to PM.

2.4. DRAM Cache / 2LM Mode

As discussed in Section 1.2, Optane DIMs can act as system memory with DRAM operating as a transparent, hardware managed, direct-maped cache. In this section, we use microbenchmarks to try to deduce the performance implications of the Cascade Lake DRAM cache design. Our results are summarized in Table 2.2 and Figure 2.3.

2.4.1. Methodology. To study the behavior of the 2LM DRAM cache, we used the same benchmarks discussed in Section 2.3 and the same methodology for measuring bandwidth. In this case, data gathered from the performance counters allows us to differentiate DRAM and PM traffic. Furthermore, the tag related performance counters in each IMC allows us to correlate tag events with memory traffic. Each IMC only allows four events types to be recorded at a time. Since our benchmarks are long running and largely deterministic, we run them twice to obtain both bandwidth and tag events.

Table 2.2 summarizes the observed actions required for each type of access to the IMC. We define two types of requests to the IMC. An *LLC Read* is a request from the LLC for data from the DRAM cache or PM. This request is generated on a load or store miss at the LLC. Stores can generate an LLC read as they may require a RFO. An *LLC Write* is a request from the LLC to write back dirty data to the DRAM cache. LLC write requests are generated either when a dirty line is evicted from the LLC or from a *nontemporal* store.

Furthermore, the hardware performance counters differentiate between three different types of cache accesses: *hit*, *clean miss*, and *dirty miss*. A *hit* implies that address accessed by an LLC request is present in DRAM. A *miss* means that an address is *not* resident in DRAM and must be fetched from PM. Since this cache is direct mapped, a miss implies that some other data is occupying the set corresponding to the requested address. A miss is *dirty* if this aliasing data has been modified since its original insertion and thus must be written back to PM upon eviction.

To study read and write hits, we use the read-only and write-only benchmarks respectively on a 51 GiB array backed by 1 GiB hugepages to mitigate TLB overheads. Because the array is far larger than the 38 MB LLC cache, each CPU load generates an LLC read and each CPU *nontemporal* store generates an LLC write. This array is also small enough to fit in the DRAM cache without aliasing. Thus, all LLC reads/writes accesses will be cache hits.

Generating clean LLC read misses and dirty LLC write misses is also straightforward. We use a 420 GB array, which is over twice the size of the 192 GB DRAM cache per socket. Applying the read-only benchmark to this array for several iterations ensures a clean LLC read misses for each CPU load. Similarly, the write-only benchmark ensures that each *nontemporal* store generates a dirty LLC write miss.

Testing dirty LLC read misses and clean LLC write misses is more complicated. For dirty LLC read misses, we first prepare the 420 GB array from before by writing to it, making the entire DRAM cache is dirty. We then perform a single iteration of the read-only kernel. Thus, each CPU load early in the iteration generate LLC reads that will be a dirty miss in the cache. As the iteration progresses, however, a larger portion of these loads become clean misses as the dirty cache is replaced by clean data. Consequently, we determine cache behavior based on data collected early in the iteration. We use a similar procedure to prime and test clean LLC write misses.

When testing the behavior of the cache, we use *nontemporal* stores when writing. This ensures that the behavior shown by the IMC is purely the result of the incoming store and not an earlier RFO. For all benchmarks, we also compute an *effective* bandwidth as seen by the application. This is obtained using the size of the array and wall clock time for each benchmark.

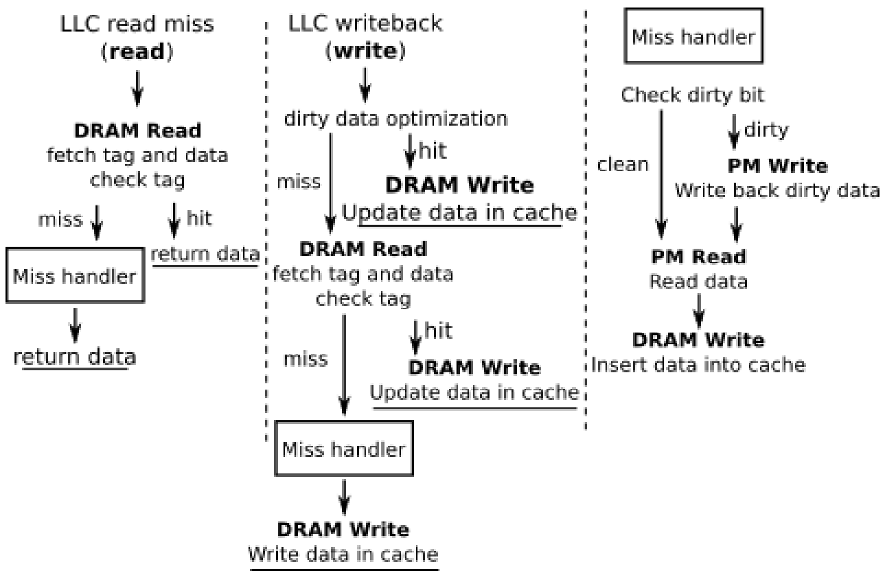


FIGURE 2.3. Flowchart showing the operation of the DRAM for LLC read misses which occur on a processor load or store which misses in the LLC and LLC write-backs which occur when a dirty block is evicted from the LLC. The miss handler is the same for reads and writes and is factored out on the right. Underlines indicate where the actions end, and bold shows the hardware actions. A summary of total memory accesses is given in Table 2.2.

While we only outlined several key benchmarks to test the different regimes of the DRAM cache, we also applied a whole range of microbenchmarks with different thread counts and access patterns to fully characterize the behavior of the cache and validate the results presented here.

2.4.2. 2LM Observations. Table 2.2 summarizes our findings for the cache events and Figure 2.3 demonstrates a flow chart of IMC logic that models this behavior. We describe each of these columns in turn. To help with our discussion, we use the term **access amplification** [**62**] as the ratio of *memory* accesses (i.e., both DRAM and PM) to demand accesses.

LLC read hits are simple. The IMC initiates a DRAM read, which fetches data along with the tag in the ECC bits. A tag check is performed and since the tag matches, the data is immediately forwarded with no access amplification.

Figure 2.4a shows bandwidth for the read-only benchmark in the 100% clean miss scenario. Note a 3× access amplification for each miss. Essentially, the tag miss is serviced by a miss handler, which fetches the requested cache line from PM, inserts into DRAM, and forwards to the CPU.

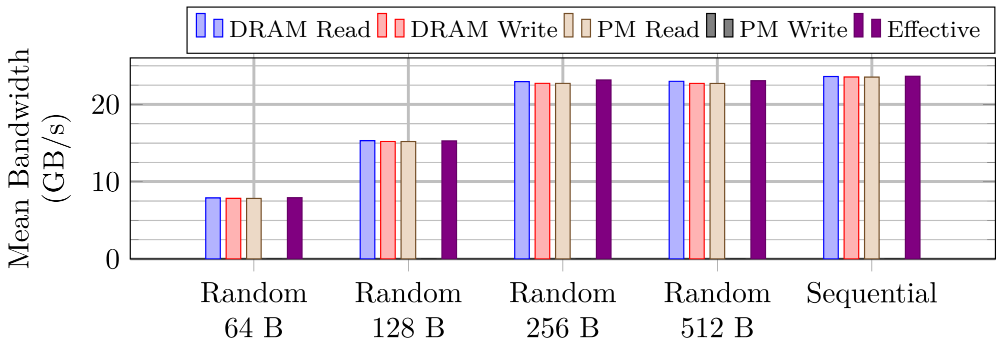
Dirty read misses are handled much the same as clean read misses. The only change is that the cache line evicted from DRAM must be written back to PM.

LLC write hits incur a 2× access amplification because the IMC must first emit a DRAM read to perform a tag check. Only upon verification of the tag can the line be safely written.

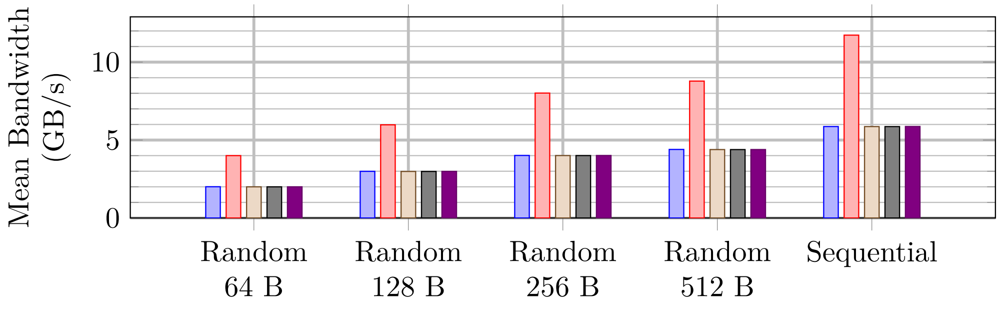
Next, we discuss dirty LLC write misses. Figure 2.4b shows collected bandwidth for the write-only benchmark where each *nontemporal* store is a dirty tag miss. Observe a 2× access amplification in DRAM writes alone. Upon receiving a completely dirty cache line store yielding a tag miss, we would expect the IMC to write the evicted line to PM and directly insert the incoming line to DRAM. This would yield a total of 1 DRAM read (for the tag check), 1 PM write, and 1 DRAM write. However, the data in Figure 2.4b suggests that this is not the case. Our best guess is that the memory controller *always* inserts on a miss (regardless of whether that miss was a read or write). The second DRAM write is thus the actual write of cache line to DRAM. Clean LLC write misses are similar dirty write misses without the PM write back.

2.4.3. Dirty Data Optimization. Finally, this brings us to the phenomenon that we call the Dirty Data Optimization (DDO). At times, the memory controller is able to elide the tag check (i.e. DRAM read) and instead directly forward LLC writes to DRAM. This can be seen in Figure 2.4c which shows the distribution of traffic for the read-modify-write benchmark in a 100% dirty LLC miss scenario using *standard* stores. The CPU load initiates a dirty LLC read miss (dirty from a previous write), accounting for one DRAM read (tag check) plus the traffic associated with a cache insert. Since *standard* stores are used, the subsequent CPU store will remain in the LLC for some time before being evicted and written to memory. Thus, there is low temporal locality between a cache line's load and its write back.

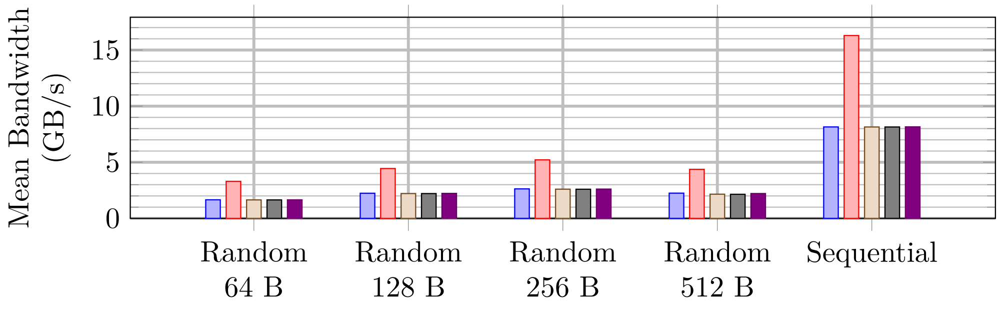
Due to this low locality, we would *expect* this delayed LLC write to require another tag check, resulting in a total of two DRAM reads per CPU load-store pair. However, this is not the case and it appears this second tag check is elided. While this could be explained by an inclusive cache, we found that this is not the case as it is possible to have small amounts (< 8 KiB) of aliasing data simultaneously within the CPU cache. Thus, we are not sure the exact mechanism driving this optimization.



(a) *Read-only* benchmark, clean LLC read misses, 24 threads.



(b) *Write-only* benchmark, dirty LLC write misses, 24 threads, *nontemporal* stores. Using 4 threads only increases the maximum write bandwidth by 1 GB/s.



(c) *Read-modify-write* benchmark, dirty LLC read miss followed by a later DDO LLC write, 4 threads, *standard* stores. Sequential achieves the highest PM write bandwidth of any 2LM benchmark with neglibile difference between *nontemporal* and *standard* stores.

FIGURE 2.4. Benchmark results on a large array exceeding the size of the DRAM cache. Because the array size exceeds DRAM, the miss rate in the DRAM cache is 100%. The "effective" bar illustrates performance as seen by the application, computed by wall clock time and data accessed.

2.4.4. Discussion. We described our observation of 2LM's mechanics, but what does this mean for user applications? There are two points we want to make. First, contrast Figure 2.4, which shows the effective NVDIMM bandwidth in 2LM with a high miss rate, with Figures 2.2a and

2.2b, showing the maximum speed of PM. The highest PM read bandwidth in 2LM (Figure 2.4a) is 23 GB/s and the highest write bandwidth (Figure 2.4b) is 8 GB/s. This is 60% and 72% the demonstrated achievable bandwidth of our system's PM. This is the ideal case with well formed traffic. We expect applications with a large memory footprint (exactly those that would benefit from the large memory pool provided by PM) and a high DRAM cache miss rate to experience a severe bandwidth bottleneck. Second, cache misses are costly in terms of extra traffic generated, with LLC read and write misses generating up to 3× and 5× access amplification. This is costly both in terms of energy and lost bandwidth.

So far, we have demonstrated the potential for applications to experience bandwidth bottlenecks in 2LM. In the next two sections, we provide case studies demonstrating this effect on real applications.

2.5. Case Study 1: Convolutional Neural Networks

In this section, we will take a deep dive into some of pitfalls a bandwidth and compute heavy application can fall into when running under 2LM. Specifically, we consider the problem of training deep Convolutional Neural Networks (CNNs) whose working set size greatly exceeds the physical DRAM of a system, requiring the extra memory provided by PM.

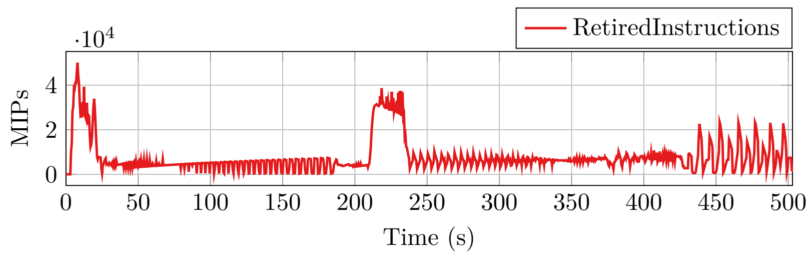
CNNs are typically expressed as a directed acyclic graph of computation primitives such as convolutions and matrix multipications, that are heavy on compute, and operations such as batch normalization and concatenation that are heavy on bandwidth requirements. At a high level, a single *iteration* of training consists of a *forward* pass, during which the network is evaluated (almost) normally on a batch of training data (some kernels like Batch Normalization have slightly different versions for training and inference [**47**]). The output of the forward pass is compared to an expected output to generate a loss value, which is used in the backward pass to compute the partial derivative of the loss with respect to each of the trainable parameters of the network. The parameters of the network are adjusted based on the these derivatives. An important aspect of the backpropagation algorithm is that many intermediate values computed during the forward pass must be preserved to compute the backward pass. Thus, the active memory footprint of the network during an iteration of training increases during the forward pass, then decreases during the backward pass. It takes many such iterations of training across different input samples to fully train a CNN.

2.5.1. Methodology. We implemented three popular large CNNs: Inception v4 [**100**], Resnet 200 [**41**], and DenseNet [**46**] using the *ngraph* compiler [**24**] on the PM-based system described earlier. Intel's *ngraph* compiler is an optimizing compiler specifically targeting static deep neural networks that takes advantage of the Xeon CPU ISA. For these large networks, we scaled the training batch size until the overall footprint of these applications exceeded 650 GB, well beyond the capacity of the DRAM cache. All networks were run on a single NUMA node and assigned all 24 physical cores on that node with no hyper-threading. These networks were run for two warm up iterations to trigger on-demand paging by the OS and to prepare the state of the DRAM cache.

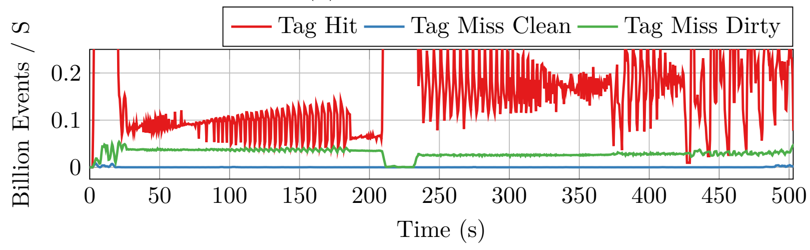
During the execution of these networks, we sampled hardware performance counters for bandwidth and tag statistics. Furthermore, we modified the *ngraph* compiler in two ways. First, we added an option to emit high resolution timestamps when beginning the execution of each compute kernel, allowing us to correlate these events with performance counter data. Second, we exposed information regarding the memory assignment of intermediate tensors. This allows us to examine which regions of memory are being accessed throughout the network execution.

2.5.2. Results. For the deep dive, we present the results for DenseNet [**46**], a CNN with a complicated dataflow pattern. In Figure 2.5 we break down the bottlenecks of a single iteration of training for DenseNet 264 with batchsize 3072. The baseline memory footprint for this application is around **688 GB**. Figure 2.5a demonstrates the system's retired instruction rate through time. Figure 2.5b shows the number of tag hits, dirty tag misses, and clean tag misses throughout the iteration.

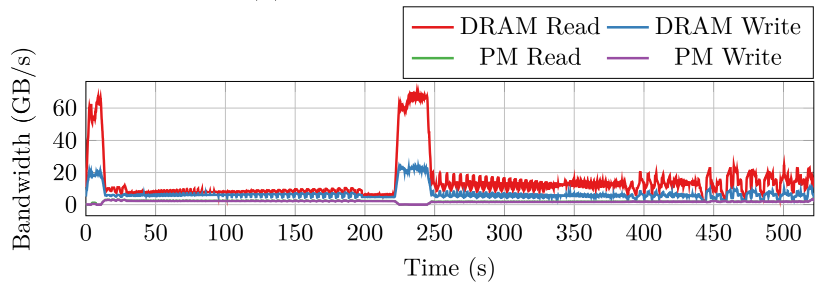
Key observations to make are: (1) there very few clean tag misses, (2) there is a high percentage of dirty tag misses, both in the forward pass and the backward pass, and (3) there noticeable regions of high tag hits at the beginning of the forward and backward passes with a corresponding drop in dirty tag misses. Finally, Figure 2.5c breaks down the read and write bandwidths to DRAM and PM. Regions of high dirty miss rate correspond to low bandwidth and instruction throughput. Reasonable system performance is only achieved when the hit rate is high.



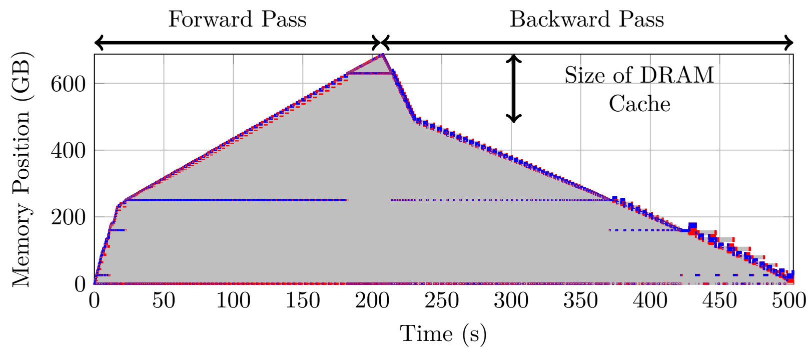
(a) System MIPS.



(b) DRAM cache statistics.



(c) Memory bandwidth through time. PM read and write bandwidths are similar, thus the PM read line is hidden behind the PM write line.



(d) Live memory in the *ngraph* heap. Memory that is highlighted *gray* indicates memory that will be read before written (i.e., live memory). *Blue* indicates a write is happening. *Red* indicates a read is happening. *White* shows memory that will be written before read.

FIGURE 2.5. Memory behavior of a single iteration of training for DenseNet 264 with a batchsize of 3072.

So, a good question at this point is - *Why are so many dirty tag misses generated, and why are there regions of high cache hit rate?* Two related phenomena can explain this.

Figure 2.5d shows the memory usage of DenseNet through time for a single iteration of training. Before execution, the *ngraph* compiler allocates a single buffer for the *entire* network. The offset from the base of this buffer is shown on the vertical axis of Figure 2.5d. The change in memory state through time is shown using different colors. The color *white* indicates that the region of memory is **free** (semantically speaking). That is, it will always be written to before it is read by the program. A *blue* highlight indicates that a region of memory is being actively written to, *red* indicates a read, and *gray* indicates that the memory will be read from in the future.

For an iteration of training, first the *forward* pass of the model is computed (up to time around 220, annotated in Figure 2.5d). Throughout the forward pass, some of the generated intermediate tensors must be held in memory to facilitate computation of the backward pass. Thus, the amount of live memory (gray) accumulates through the forward pass. Once a preserved tensor is used on the backward pass, the region in memory where it was stored is free for further use (white). The *ngraph* compiler takes advantage of this newly freed area to allocate intermediate tensors required to compute the backward pass. This is the very subtle streak of blue on the right shoulder of Figure 2.5d.

However, from the perspective of the 2LM cache, the fact that writes are occurring to a region of memory on the backward pass makes memory is dirty with respect to the DRAM cache. Hence, even when this region of memory is semantically free from the program's perspective, the cache must still generate a dirty write back upon eviction. *Because the DRAM cache is unaware of the meaningful lifetime of memory, it generates a large amount of unnecessary traffic.*

Finally, the regions of high DRAM cache hit rate occur at the beginning of the forward and backward pass because the area of active memory folds back on itself. Recent data is in the cache, so all accesses are cache hits. This continues until the entire cache has been read, at which point further accesses are cache misses.

2.5.3. Problematic Kernels. To wrap up this section, we will explain the relatively high frequency periodic behavior that is noticeable in the Tag Hit line of Figure 2.5b. DenseNet is composed of a linear chain of "dense blocks" where each dense block consists of a sequence of

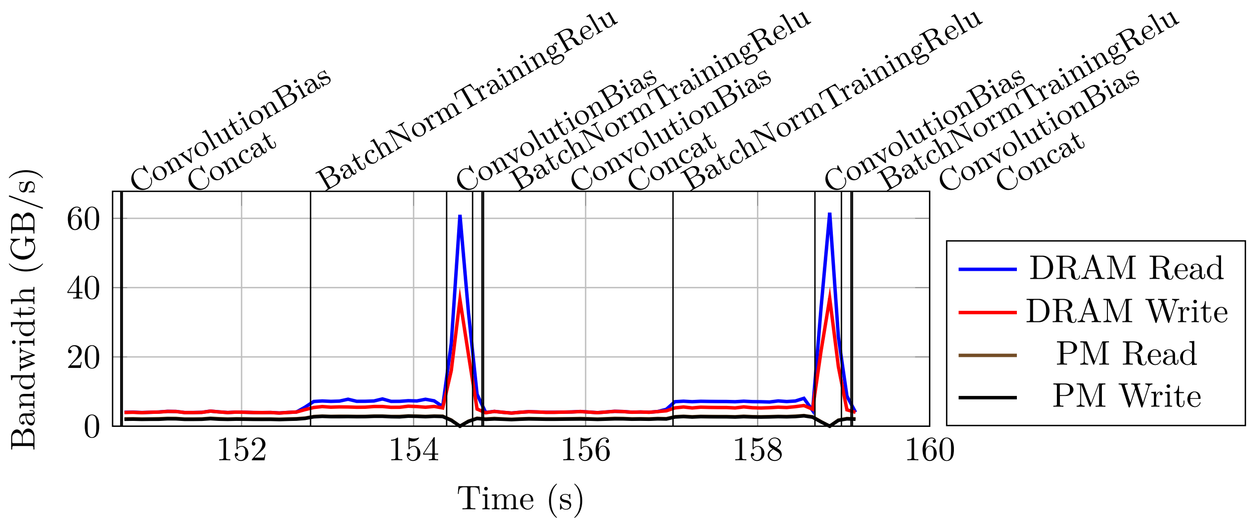


FIGURE 2.6. Snapshot of periodic bandwidth behavior during the forward pass of training DenseNet 264 in 2LM. Vertical bars mark the start of kernel execution. Very short running kernels have been excluded for clarity.

Concat, BatchNorm, Conv, BatchNorm, and Conv operators. Figure 2.6 shows a high resolution snapshot of the bandwidth for two such dense blocks during the forward pass of DenseNet. The point where kernels begin execution is annotated on the graph. The main performance bottlenecks apparent in Figure 2.6 are Concat and BatchNorm. These are both memory-bound kernels with little data reuse and are more affected by the low bandwidth associated with a high dirty tag miss rate. The second BatchNorm within each dense block operates on much smaller intermediate tensors, and is thus less impactful on overall performance. Similar problematic kernels exist on the backwards pass as well, including BatchNormBackprop and the back-propagation kernels for the filter/bias inputs of $3x3$ convolutions.

2.5.4. Discussion. In summary, the overall performance of CNN training in 2LM mode in PM-based systems is affected by two factors: (1) low effective bandwidth with a high miss rate and (2) a significant amount of unnecessary dirty writebacks. From the microbenchmarks, the first of these is not too surprising. However, the second exposes a performance pathology *not* demonstrated by the microbenchmarks, made worse by the relatively low write bandwidth of PM. Next, we will look at a different class of algorithms that suffer similarly.

2.6. Case Study 2: Graph Processing

In this section, we perform a preliminary study on applications known for having diverse performance characteristics and irregular memory access patterns. To accomplish this, we evaluate a variety of graph processing algorithms on large real world graph inputs using Galois [**68**], a high performance shared memory graph analytics framework.

2.6.1. Background. Large graph processing has garnered substantial research interest across a variety of use cases, including the identification of social media influencers and decision makers, or finding fraudulent actors within a business network. These real world large systems require frameworks process representative graphs with tens of billions of nodes and trillions of edges, incurring a high memory footprint that is expensive to accommodate in DRAM. Depending on the topology of the input graph and the processing algorithm being used, the memory access pattern can vary wildly. This presents a challenge when optimizing such workloads for systems with limited main memory.

To address this issues, several efforts [**26**, **33**] have explored leveraging PM for graph analytics on a single machine. However, such works focused on performing an analysis and comparison of different graph processing frameworks and system settings to optimize the use of Optane for graph workloads. Here, we evaluate the bandwidth characteristics of such irregular workloads in 2LM.

2.6.2. Methodology. Graph kernel experiments were run on the shared memory graph analytics framework Galois. Specifically, our evaluations consisted of 4 benchmarks from the lonestar suite: breadth-first search (bfs) [**20**], connected components (cc) [**93**, **96**], k-core decomposition (kcore) [**25**], and pagerank-push (pr) [**73**]. These kernels were chosen based on their diverse execution characteristics [**6**]. Our workloads were run with the settings by Gill et al. [**33**]. For *bfs*, the source node was the maximum out-degree node. The tolerance of pr was set to 10^{-6} and we used the $k = 100$ for kcore. Each kernel ran until convergence, except for pr which ran for 100 rounds.

We used two realistic unweighted massive input graphs: wdc12 [**72**], the largest publicly available graph, and kron30 [**60**], a randomized scale free graph generated using a graph500 based kronecker generator [**34**]. Each were chosen to highlight the differences between when a graph fit and did not fit in the DRAM cache. While these graphs have different structures, we can still draw conclusions from kernels' relative performance on these graphs. Both were processed using the provided graph-converter in Galois and resulted in binaries of size 507 GB and 73 GB respectively.

In 2LM, all benchmarks were run on two NUMA nodes and assigned all 96 threads. Since two sockets are used, the size of the DRAM cache is effectively doubled to 384 GB with 6 TB of PM. The total NUMA interleaving and 2 MiB hugepages were used with no page migration to maximize performance [**33**].

To find the baseline data movement required by the algorithms, we configured the PM regions on each socket as extra NUMA nodes. This is facilitated through the `daxctl`[3] tool with the machine in 1LM. Since Galois uses a NUMA preferred policy, the threads on each socket will initially allocate memory on that socket's DRAM. When DRAM is exhausted, further allocations are serviced by PM. By summing the traffic to DRAM and PM, we can establish the baseline memory traffic required by each application.

As with our previous experiments, measurements on bandwidth and tag statistics were gathered using hardware performance counters.

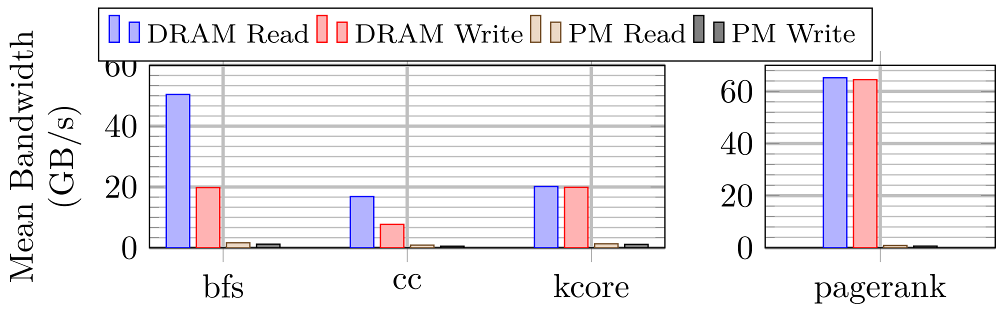
2.6.3. Results. Figure 2.7 compares the observed bandwidth when running the graph kernels on *kron30* and *wdc12*. When processing *kron30*, the kernels have a working set that largely fits within the DRAM cache while the working set when processing *wdc12* greatly exceeds the DRAM cache. When the working set does not fit in the DRAM cache, there is a significant decrease in DRAM bandwidth during an algorithm's execution.

Figure 2.8 shows the total amount of data moved in the NUMA and 2LM configurations for PM. Since page migration was disabled, Figure 2.8a shows the true demand accesses of the workload. Comparing this with Figure 2.8b we see significant access amplification.

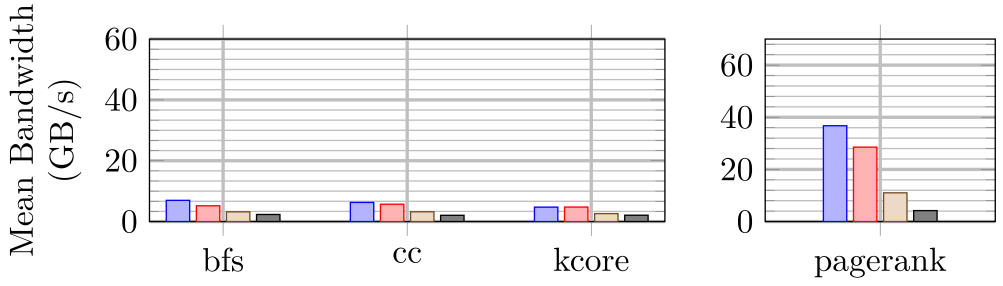
Figure 2.9 shows the workload characteristics of the pagerank-push algorithm for both *kron30* and *wdc12*. Figure 2.9a shows the algorithm's bandwidth when its working set largely fits in the cache. Bandwidth is stable at 70 GB/s with roughly equal DRAM reads and writes.

On the other hand, Figure 2.9b demonstrates the bandwidth of pagerank-push when its working set *does not* fit in the DRAM cache. Not only is the average bandwidth significantly lower, but there is also an excess of DRAM reads coupled with heavy PM traffic. The tag metrics shown in

[3]`https://docs.pmem.io/ndctl-user-guide/daxctl-man-pages`

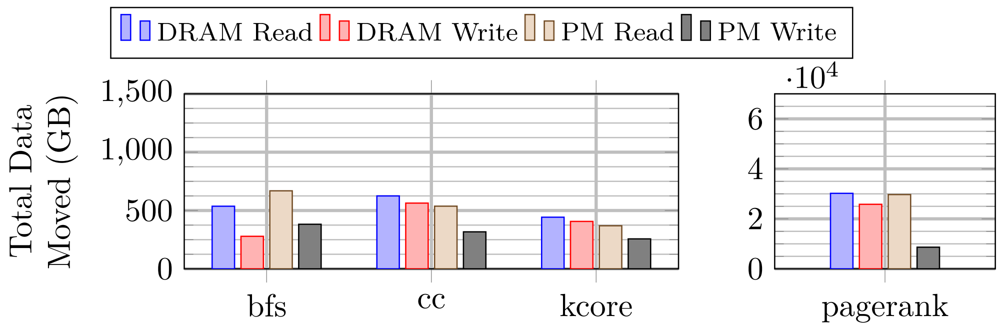


(a) Performance of graph kernels on *kron30* which fits in DRAM

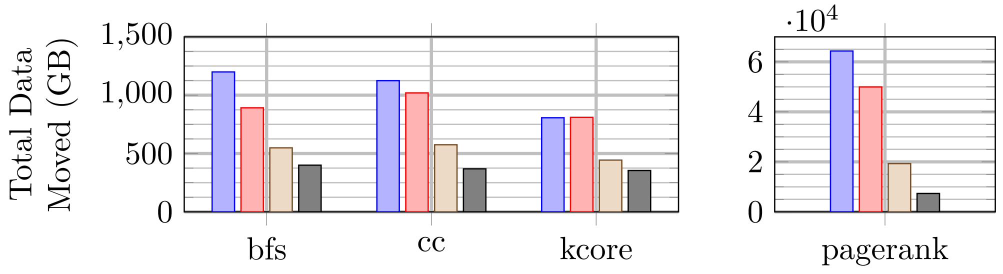


(b) Performance of graph kernels on *wdc12* which exceeds DRAM capacity

FIGURE 2.7. Graph kernel performance in 2LM run on 96 threads. When the input graph does not fit in the DRAM cache, bandwidth significantly drops.

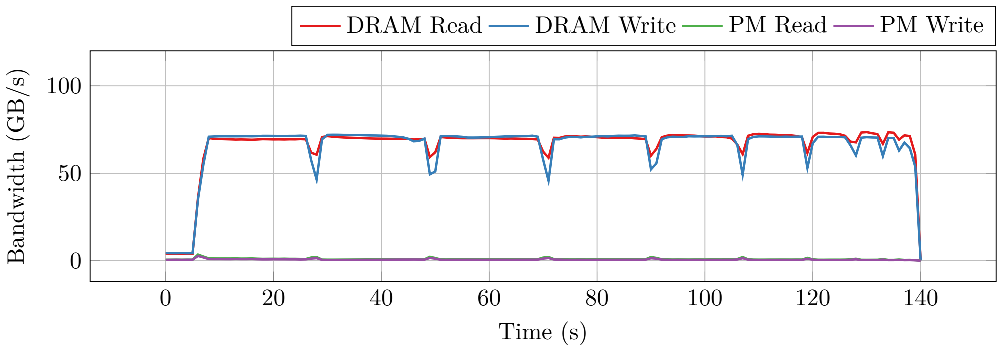


(a) PM as extra NUMA nodes.

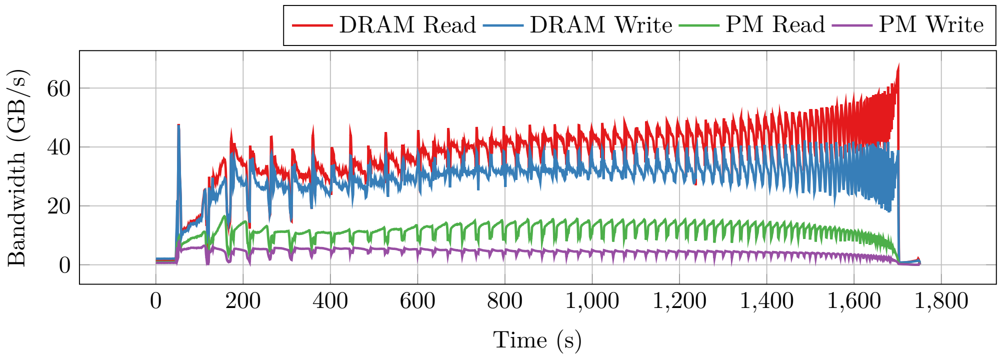


(b) PM as system memory with a DRAM cache.

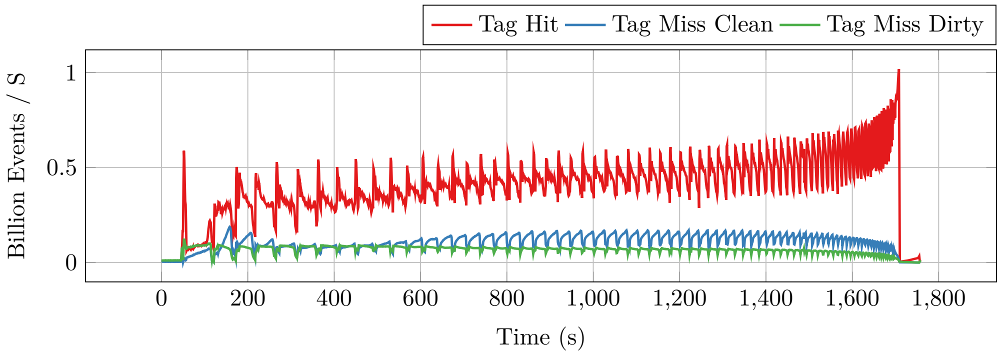
FIGURE 2.8. Total amount of data moved during the execution of a graph kernel when the input graph does not fit in the DRAM cache.



(a) Bandwidth trace for *kron30*, which largely fits within the DRAM cache.



(b) Bandwidth trace for *wdc12*, which greatly exceeds the capcity of the the DRAM cache.



(c) Tag trace for *wdc12*.

FIGURE 2.9. Traces for the *pagerank-push* algorithm. Figure 2.9a demonstrates behavior when the graph largely fits within the DRAM cache. Conversely, Figures 2.9b and 2.9c shows behavior when the working set greatly exceeds the DRAM cache.

Figure 2.9c show the presence of both clean and dirty tag misses as well as the correlation between hit rate and DRAM bandwidth.

2.6.4. Discussion. As with CNN training, large scale graph processing is a workload with a high DRAM cache miss rate. This is made worse since traditional graph algorithm implementations involve mutating the in-memory representation of the graph [**33**]. In 2LM, this mutation will mark the corresponding memory as dirty. Thus, not only is the miss rate high, but many of these misses require PM write backs, which we have demonstrated to be inefficient. As a result, it is not surprising that 2LM behaves poorly for these particular implementations.

2.7. Discussion and Mitigation Strategies

In this chapter, we demonstrated that the DRAM cache as currently implemented in Intel's Cascade Lake systems performs poorly for applications with a high miss rate. We showed that a DRAM cache miss can cause 3–5× more memory accesses than the original demand requests. Further, we showed that this causes performance degradation in two bandwidth-limited workloads: CNN training and graph analytics which are important use cases for PM since they have extremely large memory footprints. Furthermore, we show that certain data reuse semantics at the program level can cause severe degradation.

For instance, in the deep neural network training workload, a significant amount of the data movement from the DRAM cache to PM is *useless* as this data was only meant to be used temporarily by the program and will be overwritten before it is read again. This dirty temporary data dominates the DRAM cache leading to more misses than necessary and limiting performance to the smaller PM write bandwidth.

2.7.1. Software-managed multi-level memory. So what can be done about this? In this section, we look briefly at an example of software-managed memory for graph analytics. We propose that through software-managed memory, better performance can be obtained than using the hardware-managed cache in 2LM mode for these miss heavy bandwidth-bound workloads. In the next chapter, we will look at software management techniques for tackling memory management for CNN training.

Software management relies on decoupling the DRAM and PM memory pools. So far, this chapter focused on the 2LM (or "memory mode") of the PM systems, these systems can also be configured in “app-direct mode” or 1LM where the programmer has full control over the data location and movement. PM is simply mapped into a program’s address space.

2.7.1.1. *Graph Analytics.* As pointed out in Section 2.6, graph algorithm implementations in Galois and other graph frameworks often mutate graph data structure. With PM, this is an issue due its low write bandwidth (which is further exacerbated by 2LM’s write amplification). To tackle this issue, the authors of Sage [**33**] designed that software specifically with PM in mind. Their key approach is to (as much as possible) use PM for read only data.

When running algorithms that require tracking state (such a nodes visited for bfs), an auxiliary DRAM-based data structure is used. This data structure is greatly compressed and supplements the read-only PM-based adjacency list. Mutation is only performed on the auxiliary data structure, and hence write traffic is only generated to DRAM. To optimize for multiple sockets, Sage takes advantage of PM’s capacity to keep a full copy of the graph on both CPU sockets. With these techniques, they were able to design algorithms 1.87× faster on average than GBBS and 1.94× faster on average than Galois in 2LM [**33**].

This is an example demonstrating that clever software management can over come the bandwidth limitations of PM. Conversely, these same limitations are exacerbated by access amplification caused by the DRAM cache.

2.7.1.2. *Techniques for CNNs.* In the next chapter, we will present AutoTM, a heterogeneous memory management technique for CNNs. Unlike Sage’s approach to graph analytics, CNN’s cannot use PM as just read-only memory. Thus, AutoTM uses mathematical optimization to determine where to place intermediate data and when to move this data between memory pools.

CHAPTER 3

Compiler-Based Heterogeneous Memory Management for Statically Analyzable Workloads

3.1. Introduction

Deep Neural Networks (DNNs) have been dramatically successful over the past decade across many domains including computer vision [**57**], machine translation and language modeling [**99**], recommendation systems [**67**], speech [**110**] and image synthesis [**111**], and real-time strategy game control [**105**]. This success has in turn led practitioners to pursue larger, more expressive models. Today, state of the art models in language modeling and translation have 100s of billions of parameters [**94**] which requires 100s of GB of active working memory for training. For instance, large models such as BigGAN [**9**] found significant benefits from increasing both model size and training batch size, and Facebook's recent DLRM recommendation system [**67**] contains orders of magnitude more parameters than conventional networks. Additionally, to reach beyond human-level accuracy these models are expected to grow even larger with possibly 100× more parameters [**42**]. The large memory footprints of these models limits training to systems with large amounts of DRAM which incur high costs.

As the memory capacity demands of DNN training are growing, new high density memory devices are finally being produced. Specifically, Intel® Optane™ DC Persistent Memory Modules (PM) [**33**, **48**] can now be purchased and are up to 2.1× lower price per capacity than DRAM. These devices are on the main memory bus, allowing applications direct access via load and store instructions and can be used as working memory. Thus, in this chapter we ask the question "*what are the design tradeoffs of using PM in training large DNN models, and more specifically, can PM be used as a DRAM replacement when training for large DNN models?*"

Figure 3.1 shows the training performance for three different memory systems: an all PM system (lowest cost), an all DRAM system (highest cost), and a heterogeneous system (moderate

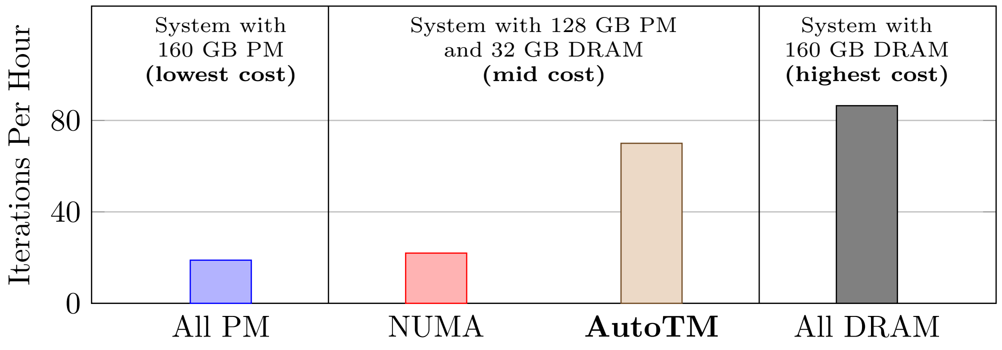


FIGURE 3.1. Performance of Inception v4. Batch size of 1472.

cost). The all PM bar shows that naively replacing DRAM with PM results in poor performance (about 5× slowdown) for training large DNN models. The first-touch NUMA [**56**] bar shows that current system support for heterogeneous memory is lacking, providing only a small benefit over the all PM case. However, AutoTM provides 3.7× speedup over the PM case and is within 20% of the all DRAM system. Thus, we find that a small fraction of DRAM reduces the performance gap between PM and DRAM, but only if we use *smart data movement*.

Use of heterogeneous memory to reduce DRAM has been studied in the past. Facebook has used SSDs to reduce the DRAM footprint of databases [**30**]. Bandana [**31**] uses SSD based persistent memory to store deep learning embedding tables [**19**] with DRAM as a small software cache. In the context of machine learning, vDNN [**89**], moDNN [**15**], and SuperNeurons [**106**] develope system-specific heuristics to tackle heterogeneous memory management between the GPU and CPU to overcome the low memory capacity of GPUs. Furthermore, future HPC systems will be increasingly heterogeneous with DRAM, PM, and HBM [**82**], so we need a solution that is general and automatic.

In this chapter we introduce AutoTM—a framework to automatically move DNN training data (tensors) between heterogeneous memory devices. AutoTM enables training models with 100s of billions of parameters and/or with large batch sizes efficiently on a single machine. We exploit the static nature of DNN training computation graphs to develop an Integer Linear Programming (ILP) [**91**] formulation which takes a profile driven approach to automatically optimize the location and movement of intermediate tensors between DRAM and PM given a DRAM capacity constraint.

We evaluate the effectiveness of AutoTM on a real system with Optane PM by implementing our approach in the nGraph compiler [**24**]. Our experiments show that naive use of PM is not effective, but intelligent use of PM and DRAM is required. Furthermore, using initial public pricing information, we evaluate the cost-performance benefits DRAM-PM based systems. We show that ratios of 8 : 1 or 4 : 1 of PM to DRAM can be more cost effective than only DRAM or only PM.

We also compare our approach to the existing hardware DRAM cache implemented in current Intel platforms [**48**] and find AutoTM offers up to 2× performance improvement over hardware-managed caching.

Finally, we demonstrate that AutoTM can be further generalized beyond PM-DRAM heterogeneity by applying AutoTM to CPU-GPU systems. The approach taken by AutoTM uses minimal problem specific heuristics and is thus a general approach toward memory management for many different heterogeneous systems.

The chapter is organized as follows. In Section 2 we present a quick overview of training deep neural networks and Intel's Optane DC PM. In Section 3 we will present the details of AutoTM and in Section 4 we will describe implementation details, followed by our evaluation methodology in Section 5, and the main results in Section 6. We will present extensions to AutoTM in Section 7 and conclude with related work and directions for future work.

3.2. Background

3.2.1. Deep Learning Training. Deep neural networks (DNNs) are often trained using a backward propagation algorithm [**58**] and an optimizer such as stochastic gradient descent. Popular deep learning frameworks such as Tensorflow [**1**] and nGraph [**24**] implement DNNs as a computation graph where each vertex or node in the computation graph represent some computational **kernel**. Common kernels include convolutions (CONV), pooling (POOL), matrix multiplication, and recurrent cells such as LSTM or GRU. Each kernel has its own characteristics such as number of inputs, number of outputs, computation time, and computational complexity. Directed edges in the computation graph between kernels denote data or control dependencies between kernels. An edge representing a data dependency is associated with a **tensor**, which we consider to be a contiguous region of memory with a known size.

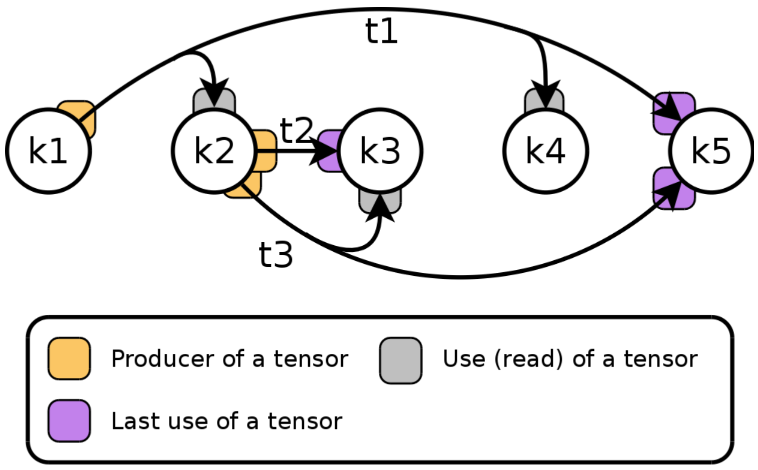


FIGURE 3.2. A simple example of a computation graph. The k nodes are the compute kernels in the graph and t edges (tensors) show the data dependency between kernels. Intermediate tensors have a finite live range that can be exploited to reduce the memory footprint of the computation graph.

Figure 3.2 shows a simple example computation graph with 5 kernels and 3 tensors. Nodes in the graph are compute kernels, each with zero or more inputs and outputs. The inputs and outputs of a kernel are immutable tensors. Each tensor is annotated with its producing kernel, each user of the tensor, and the last user of the tensor. After its last use, a tensor's memory may be freed for future tensors.

We focus on the case where the computation graph describing the training iteration is static. That is, the computation graph contains no data-dependent control behavior and the sizes of all intermediate data is known statically at compile time. While many DNN graphs can be expressed statically, there are some networks that exhibit data-dependent behavior [**94**]. In Chapter 4, we will develop techniques suitable for dynamic computation graphs.

3.2.2. Intel Optane DC PM. As discussed previously, there are two operating modes for Optane DC PM. In **2 Level Mode** (2LM or *cached*) PM act as system memory with DRAM as a direct mapped cache. This operating mode allows for transparent use of the PM at the overhead of maintaining a DRAM cache. **App Direct Mode** allows users manage the PM directly. The PM are mounted on a system as direct access file systems. Files on the PM devices are then memory mapped into an application. When using a direct access aware file system, loads and stores to addresses in this memory mapped file go directly to the underlying PM. Note that in App Direct

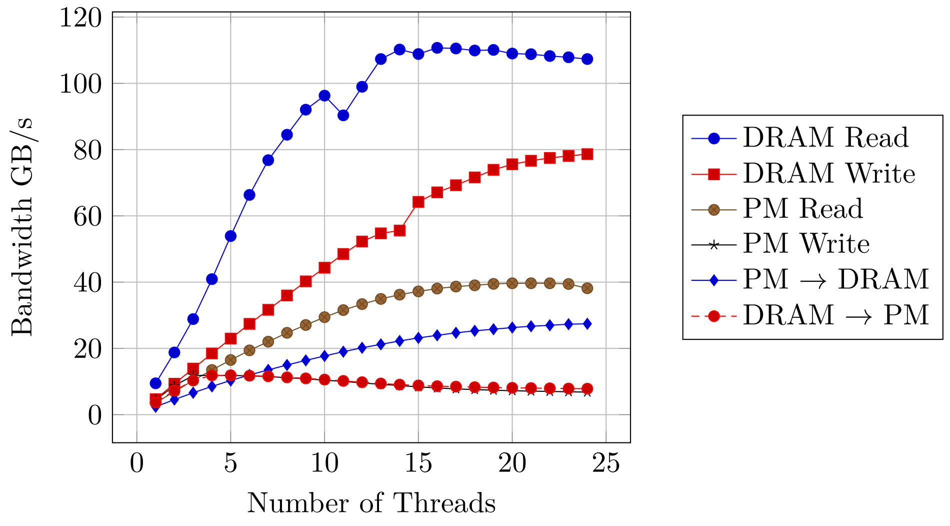


FIGURE 3.3. Read and write bandwidths between DRAM and PM. All operations were performed using AVX512 load and stores. Copies between DRAM and PM were done using streaming load and store intrinsics.

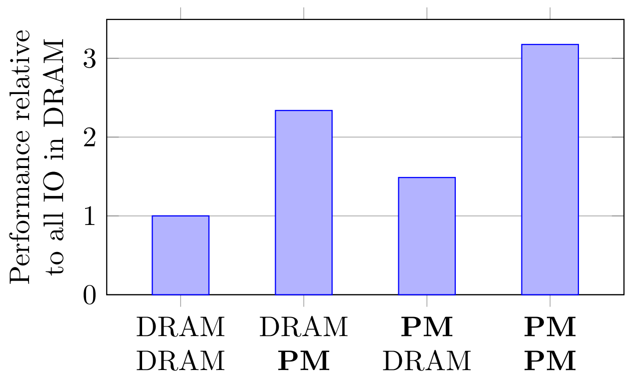


FIGURE 3.4. Execution time of a CONV kernel with input (upper label) and output (lower label) feature maps varied between DRAM and PM. The performance of the kernel is largely unaffected by the location of the output feature map. The CONV kernel had a filter size $(3, 3, 128, 128)$ and a input feature map size of $(112, 112, 128, 16)$ and was executed using 24 threads.

mode, the total available memory is the sum of DRAM and PM while in 2LM only the PM capacity is counted. In this work, we focus on using the PM in App Direct mode, and make comparisons between our optimized data movement and 2LM.

Figure 3.3 shows the read, write, and copy bandwidth of DRAM and PM on our test system with six interleaved 128 GB PM. The read, write, and copy operations were implemented by splitting a region of memory into contiguous blocks and assigning a thread to each chunk. AVX-512 streaming

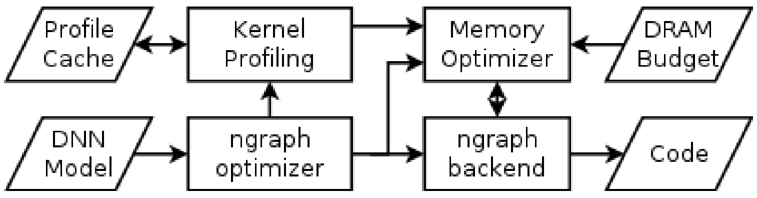


FIGURE 3.5. System Overview.

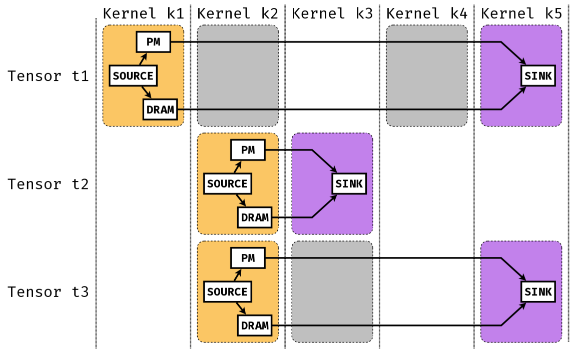
loads and stores were used to implement the copy operation as they provide significantly higher throughput between DRAM and PM.

From Figure 3.3, we make the following observations about PM bandwidth is significantly lower than DRAM, read bandwidth scales with the number of threads, write bandwidth peaks at a low number of threads and diminishes with a higher number of threads, copy bandwidth from DRAM to PM scales with the number of threads, copy bandwidth is chiefly limited by PM write bandwidth, and there is significant read/write asymmetry. These findings agree with the performance evaluation of Optane PM presented in Chapter 2.

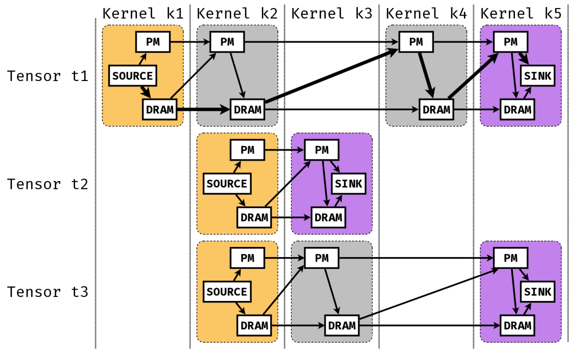
The read/write asymmetry has implications on the performance of kernels with inputs and outputs in PM or DRAM. Figure 3.4 demonstrates the performance impact on a single CONV kernel. We observe that when the input to the CONV kernel is in PM and the output is in DRAM, the performance of the kernel is comparable to when both input and output are in DRAM. However, in the cases where the output is in PM, the kernel runs over two times slower. Any system seeking an optimal runtime with a memory constraint must take these relative timings into consideration when making decision on where to assign data. In the next section, we will describe the details of AutoTM and how it manages these performance characteristics.

3.3. AutoTM

An overview of the proposed framework is shown in Figure 3.5. A DNN model is given to nGraph, which optimizes the network DAG according to the selected backend (e.g. CPU, GPU etc.). As part of the compilation process, our system inspects the nGraph DAG data structure to extract (1) the order and types nodes in the graph, (2) the tensors produced and consumed by each node, and (3) the specific kernels chosen by nGraph.



(a) Static assignment. Tensors are created into either PM or DRAM and stay there.



(b) Synchronous Movement. Tensors are allowed to move just before or just after the kernels that use or produce them. This movement blocks program execution.

FIGURE 3.6. Overlap of multiple tensor graphs and their interactions with kernels. Following the color coordination in Figure 3.2, **orange** denotes the producer of a tensor, **lilac** is the last user, **gray** marks the user of a tensor. We define the term **component** to refer to the subgraphs within each shaded region.

We then perform profiling on every kernel in the computation graph by varying its inputs and outputs between the different memory pools (i.e., DRAM or PM) and recording the execution time of the kernel in each configuration. Since this step is potentially time consuming and DNNs typically contain many identical kernels, we keep a software cache of profiled kernels. By keeping a profile cache, profiling for a given DNN only needs to be performed once. Profiling and DAG information is then fed into a Memory Optimizer (described in Section 3.3.1) along with a DRAM capacity constraint, that mutates the nGraph data structure with the tensor assignments and data movement nodes.

A user of this system only needs a nGraph function, which is a collection of “Node” and “Tensor” data structures describing computations and data flow of the compute graph. These functions can be created by using one of the nGraph front ends, or directly using C++. Profiling, optimization, and code generation all happen as part of the nGraph compilation process and is transparent to the user.

In the rest of this section, we first give a high level introduction to the memory optimizer. Then we present the details of the optimizer’s underlying ILP formulation.

3.3.1. Memory Optimizer. The goal of the Memory Optimizer is to *minimize* execution time by optimizing intermediate tensor movement and placement. The inputs to the optimizer are (1) the types of kernels in the computation graph in topological order, (2) the set of all valid tensor input/output locations for each kernel as well as profiled execution time for each configuration, (3) the sizes of all intermediate tensors, as well as their producers, users, and final users, (4) synchronous copy bandwidths between DRAM and PM, and (5) a DRAM limit. The output of the optimizer describes the location and date movement schedules for all intermediate tensors that will minimize the global execution time of the graph.

Since the Memory Optimizer is implemented as an ILP, we need to model tensor location and movement using integer or binary variables and constraints [**70**]. For each tensor t, we create a separate network flow graph $\mathcal{G}_t = (\mathcal{V}_t, \mathcal{E}_t)$ that traces the tensor's location during its lifetime. Examples of such graphs are given in Figure 3.6a and 3.6b. The structure of these graphs allows us to customize the semantics of possible tensor locations and movements.

Using this graph structure, we investigate two separate formulations, *static* and *synchronous*. The *static* formulation (Figure 3.6a) allows no tensor movement between memory pools. A tensor is assigned to either DRAM or PM and remains there through its lifetime. The *synchronous* formulation (Figure 3.6b) allows tensors to be moved between memory pools but blocks program execution to perform this movement. We further generalize the ILP formulation to an asynchronous formulation that allows overlap between computation and data movement in Section 3.7.

Network flow constraints [**35**] are placed on each tensor flow graph $\mathcal{G}_t$ so that flow out of the source vertex is 1, flow into the sink vertex is 1, and flow is conserved for each intermediate node. The solution to this network flow describes the movement of the tensor. For example, the bold path in Figure 3.6b implies the following schedule for tensor t_1: (1) created by kernel k_1 in DRAM, (2) remains in DRAM until the execution of kernel k_2, (3) after k_2, synchronously moved t_1 into PM, (4) prefetch t_1 into DRAM right before k_4, (5) move t_1 out of DRAM after k_4, (6) tensor t_1 is in PM for the execution of kernel k_5, (7) after k_5, tensor t_1 is no longer needed and can be freed from all memory pools.

3.3.2. Objective Function. We wish to *minimize* the execution time of the computation graph under a DRAM constraint. In our framework, computation kernels are executed sequentially.

Therefore, in the *static* formulation where there is no tensor movement, the objective function (expected execution time) is

$$\min \sum_{k \in \mathcal{K}} \rho_k \tag{3.1}$$

where $\mathcal{K}$ is the set of all kernels k in the computation graph and ρ_k is the expected execution time for kernel k. Note that ρ_k depends on the locations input and output tensor for kernel k. The selection of input and output tensor locations is not trivial because of dependencies between kernels. For example, if tensor t_3 in Figure 3.2 is assigned to PM, then kernel k_2 must produce t_3 into PM and kernels k_3 and k_4 must reference t_3 in PM, which has a performance impact.

Given the lower performance of PM relative to DRAM, the cost of moving a tensor from DRAM to PM may be amortized by a resulting faster kernel execution. In the *synchronous* formulation, tensor movement that blocks computation graph execution and may only happen between kernel executions. The objective function then becomes

$$\min \sum_{k \in \mathcal{K}} \rho_k + \sum_{t \in \mathcal{T}} M_t^{\text{sync}} \tag{3.2}$$

where $\mathcal{T}$ is the set of all intermediate tensors t in the computation graph and M_t^{sync} is the total amount of time spent moving tensor t. Note that a tensor t may be moved multiple times during its lifetime, so M_t^{sync} represents the sum of movement times of all individual moves of t.

3.3.3. DRAM Variables. As noted above, the execution time of a kernel depends on the locations of its input and output tensors. We must also keep track of all live tensors in DRAM to establish a constraint on the amount of DRAM used. Thus, we need machinery to describe for each kernel k whether the input and output tensors of k are in DRAM or PMEM and which tensors are in DRAM during the execution of k.

For each kernel $k \in \mathcal{K}$ and for each tensor $t \in \mathcal{T}$ where t is an input or output of k, we introduce a binary variable

$$t_{t,k}^{\texttt{DRAM}} = \begin{cases} 1 & \text{if } t \text{ is in DRAM } \textbf{during } k \\ 0 & \text{if } t \text{ is in PM } \textbf{during } k \end{cases} \tag{3.3}$$

In practice, this variable is implemented as $t_{t,k}^{\texttt{DRAM}} = 1$ if and only if **any** of the incoming edges to the DRAM node in the component in the network flow graph $\mathcal{G}_t$ for k are taken.

To determine tensor liveness, we introduce binary variables

$$t_{t,k+}^{\texttt{DRAM}} = \begin{cases} 1 & \text{if } t \text{ is in DRAM } \textbf{after} \text{ kernel } k \\ 0 & \text{if } t \text{ is in PM } \textbf{after} \text{ kernel } k \end{cases} \tag{3.4}$$

for each kernel $k \in \mathcal{K}$ and for each tensor $t \in \mathcal{T}$ where t is an output or output of k. These variables describe whether a tensor is written into DRAM after the execution of a kernel, and if it remains in DRAM until the next time it is used. In practice, this is implemented as $t_{t,k}^{\texttt{DRAM}} = 1$ if and only if the outgoing DRAM to DRAM edge is taken from the DRAM node in the component in network flow graph $\mathcal{G}_t$ for k.

We make these two distinct class of variables to handle the case in the *synchronous* formulation where a tensor is prefetched from PM to DRAM as an input to some kernel k and then moved back to PM immediately after k.

3.3.4. DRAM Constraints. Our main goal here is to establish a constraint on the amount of DRAM used by the computation graph. We must ensure that the sum of sizes of all live tensors in DRAM at any point is less than some limit $\mathcal{L}_{\text{DRAM}}$

We use the DRAM variables discussed in the previous section. First, define a helper function $\text{ref}(k, t) = k'$ where $k, k' \in \mathcal{K}$ and $t \in \mathcal{T}$ with k' defined as latest executing kernel earlier or equal to k in the topological order of the computation graph such that there exists DRAM node in $\mathcal{G}_t$ for kernel k'. For example, in Figure 3.6a, $\text{ref}(k_3, t_1) = k_1$ and in Figure 3.6b, $\text{ref}(k_3, t_1) = k_2$.

We want to ensure that at the execution time for each kernel $k \in \mathcal{K}$, the cumulative size of all live tensors resident in DRAM is with some limit $\mathcal{L}_{\text{DRAM}}$. Using the ref function, we add the following constraint for each $k \in \mathcal{K}$:

$$\sum_{t \in \mathbb{IO}(k)} |t| t_{t,k}^{\texttt{DRAM}} + \sum_{t \in \mathbb{L}(k)} |t| t_{t,\text{ref}(k)+}^{\texttt{DRAM}} \leq \mathcal{L}_{\text{DRAM},k} \tag{3.5}$$

where $|t|$ is the allocation size of tensor t in bytes, $\mathbb{IO}(k)$ is the set of input and output tensors for k, and $\mathbb{L}(k)$ is the set of all non-input and non-output tensors that are "live" during the execution of k. We assign a separate limit $\mathcal{L}_{\text{DRAM},k}$ for each kernel k initialized to $\mathcal{L}_{\text{DRAM}}$ to address the memory fragmentation issue discussed in Section 3.4.2

3.3.5. Kernel Configurations and Kernel Timing. For each kernel $k \in \mathcal{K}$, we use an integer variable ρ_k for the expected execution time of k given the locations of its input and output tensors. First, we define a configuration c as a valid assignment of each of a kernel's input and output tensors into DRAM or PM. For example, a kernel with one input and one output tensor may have up to four configurations, consisting of all combinations of its input and output in DRAM or PM.

The definition of ρ_k is then

$$\rho_k = \sum_{c \in \mathcal{C}(k)} n_{k,c} d_{k,c} \tag{3.6}$$

where $\mathcal{C}(k)$ is the set of all valid configurations c for kernel k, $n_{k,c}$ is the profiled execution time of kernel k in configuration c, and $d_{k,c}$ is a one-hot indicator with $d_{k,c} = 1$ if and only if kernel k's input and output tensors are in configuration c.

3.3.6. Tensor Movement Timing. The movement cost of a tensor t is the size of the tensor $|t|$ divided by bandwidth between memory pools. Since bandwidth may be asymmetric, we measure and apply each separately. For each tensor $t \in \mathcal{T}$, the total synchronous movement time M_t^{sync} is the sum of the number of taken edges in $\mathcal{G}_t$ from DRAM to PM multiplied by the DRAM to PM bandwidth and the number of taken synchronous edges from PM to DRAM multiplied by the PM to DRAM bandwidth.

In our case where tensors are immutable, we may apply an optimization of only producing or moving a tensor into PM once. Any future movements of this tensor into DRAM references the data that is already stored in PM. Further movements from DRAM to PM become no-ops.

3.4. Implementation Details

In this section, we describe some of the implementation details which are not directly part of the ILP formulation. The memory optimizer itself was implemented in the Julia [**7**] programming language using the JuMP [**29**] package for ILP modeling. Gurobi [**38**] was used as the backend

ILP solver. We chose nGraph [**24**] over other popular machine learning frameworks based on static computation graphs as our backend because it is optimized for the Intel hardware and is relatively easy to modify. However, AutoTM is a general technique that can be integrated into other frameworks with similar underlying semantics.[1]

3.4.1. nGraph Compiler Backend. The nGraph compiler is an optimizing graph compiler and runtime developed by Nervana Systems/Intel for deep learning (DL) applications aiming to provide an intermediate representation (IR) between DL frameworks and hardware backends. The nGraph IR is a directed acyclic graph (DAG) of stateless operations nodes, each node with zero or more inputs, outputs, and constant attributes. Inputs and outputs of each node are multidimensional arrays called tensors with an arbitrary layout. Backend kernels used to implement a node is chosen based on the attributes of the node as well as the sizes, data types, and layouts of each of its inputs and outputs. nGraph will also apply generic and backend specific whole graph optimizations such as kernel fusion and algebraic simplification.

Memory location for intermediate tensors is performed using ahead-of-time heap allocation by traversing the function DAG and maintaining a list of live tensors. When tensors are last used, the memory space occupied by those tensors is freed and used for future tensors.

3.4.2. Managing Memory Fragmentation. The ILP formulation presented thus far assumes perfect memory management, which means that if the sum of sizes of live tensors is under the memory limit, then all tensors *will* fit within memory. In practice, this is not always the case. The process of allocating and freeing tensors may fragment memory resulting in a larger memory requirement.

To manage this, we use an iterative process of reducing the DRAM limit for kernels where the the following limit is exceed and rerunning the ILP.

(1) We initialize the kernel-wise DRAM limits $\mathcal{L}_{\text{DRAM},k}$ to the $\mathcal{L}_{\text{DRAM}}$.
(2) We solve the ILP using the current values of $\mathcal{L}_{\text{DRAM},\mathcal{K}}$. nGraph translates the resulting schedule and then executes its memory allocator pass.

[1]All of the AutoTM code can be found on GitHub at **`https://github.com/darchr/AutoTM`**.

(3) We collect the set of kernels $\mathcal{K}_{\text{frag}}$ where the total amount of memory allocated exceeds $\mathcal{L}_{\text{DRAM}}$ due to fragmentation. If this set is empty, we are done.

(4) Otherwise, we apply an update $\mathcal{L}_{\text{DRAM},k} = 0.98\mathcal{L}_{\text{DRAM},k}$ for all $k \in \mathcal{K}_{\text{frag}}$ and go back to step (2).

Thus, the ILP solver may have to run multiple times before a valid solution is found. In practice, this process is usually only done 1 to 2 times with a maximum of 5 as discussed in Section 3.6.6.

3.4.3. Data Movement Implementation. Synchronous movement operations are integrated as new *move* nodes in the nGraph compiler, which are automatically inserted into the nGraph computation graph following memory optimization. The implementation of these move nodes uses a multithreaded memory copy with AVX-512 streaming load and store intrinsics followed by a fence.

Operation scheduling in nGraph consists of a simple topological sort of the nodes in the computation graph, beginning with the input parameters. This creates unnecessary memory usage with move nodes as they are scheduled ad hoc, resulting in tensor lifetimes that are longer than necessary. Thus, we extended the nGraph scheduler so that if a tensor is moved from DRAM to PM after some kernel k, we ensure that this movement occurs immediately *after* the execution of k. Conversely, if a tensor is moved from PM to DRAM to be used for kernel k, we ensure this occurs immediately *before* the execution of k.

3.5. Evaluation Methodology

3.5.1. System. Our experimental Optane DC system was a prototype dual socket Xeon Cascade-Lake server. Each socket had 6×32 GB of DRAM and 6×128 GB Intel Optane DIMMs. Each CPU had 24 hyperthreaded physical cores. In total, the system had 384 GB of DRAM and 1.5 TB NVDIMM storage.

NUMA policy was set to local by default. Unless specified otherwise, all experiments were conducted on a single socket with one thread per physical core. Each workload was run until execution time per iteration (traversal of the computation graph) was constant. Since these workloads contain no data dependent behavior, performance will be constant after the first couple of iterations. Checks were used to ensure no IEEE NaN or subnormal numbers occurred, which can have a significant impact on timing [**3**].

Our approach does not change the underlying computations performed during training; it is a transparent backend implementation optimization. Thus, the performance of our benchmarks across a few training iterations is sufficient to obtain performance metrics.

We chose to evaluate AutoTM with a multicore CPU platform because Optane PM are only available for CPU platforms. However, the ILP formulation of AutoTM should apply to any heterogeneous memory system. We explore one other example with CPU and GPU DRAM in Section 3.7.

3.5.2. DNN Benchmarks. We choose a selection of state of the art Deep Neural Networks for benchmarking our approach. A summary of the benchmarks and batch sizes used is given in Table 3.1. Conventional CNNs for the Optane DC system were Inception v4 [**100**], Resnet 200 [**41**], DenseNet 264 [**46**], and Vgg19 [**97**]. All but Vgg19 have complex dataflow patterns to stress test AutoTM. The batch sizes were chosen to provide a memory footprint of over 100 GB for each workload. These batch sizes, while larger than what is typically used, mimic future large networks while still fitting within the DRAM of a single CPU socket of our test system.

We also compare our approach against the native 2LM mode, which is a hardware solution to data management that uses PM transparently with CPU DRAM as a cache. Since we can not change the physical amount of DRAM used by 2LM, we used very large neural networks that exceed the CPU DRAM and require the use of PM to train. These very large networks include Vgg416 [**89**] (constructed by adding 20 additional convolution layers to each convolution block in Vgg16) and Inception v4 with a batch size of 6144.

3.5.3. Experiments. We want to determine whether PM is cost effective for training DNNs, and how AutoTM compares against existing solutions to use PM.

For the conventional benchmarks, we consider the impact of performance with different ratios between PM and DRAM. These ratios are given in the form $a : b$ where a is the amount of PM relative to b the amount of DRAM used to train the network. A ratio of $1 : 1$ indicates that a network was trained with half PM and half DRAM. For a network requiring 128GB total to train would have a split of 64 GB PM and 64 GB DRAM. Setting a ratio such as this may lead to a larger total memory footprint in total due to memory fragmentation in both PM and DRAM. However,

Benchmark	Batchsize	System	Baseline Memory (GB)
Inception v4	1024	PM	111
Resnet 200	512	PM	132
Vgg 19	2048	PM	143
DenseNet 264	512	PM	115
Inception v4	6144	Large PM	659
Vgg 416	320	Large PM	658
Resnet 200	2560	Large PM	651
DenseNet 264	3072	Large PM	688
Inception v4	64, 128, 256	GPU	7.6, 14.7, 29.8
Resnet 200	32, 64, 128	GPU	8.7, 16.9, 32.2
DenseNet 264	32, 64, 128	GPU	8.5, 16.8, 32.4
Vgg 19	64, 128	GPU	7.1, 12.6

TABLE 3.1. Summary of the benchmarks used in this work.

in practice the total memory footprint expansion is minimal with an observed maximum observed value of 3.83% occuring in the *static* formulation for Inception v4. A ratio of 0 : 1 denotes a system where only DRAM is used while 1 : 0 is a system using only PM.

We use a baseline of a first-touch NUMA allocation policy with DRAM as a near node and PM as a far node for the conventional benchmarks. The NUMA policy was encoded in our framework by assigning intermediate tensors to DRAM as they are created until the modeled memory capacity of DRAM is reached. Future tensors can only reside in DRAM if existing tensors are freed.

For the large benchmarks, we compare our approach to the 2LM hardware managed DRAM cache to determine the effectiveness of AutoTM relative to an existing approach.

3.6. Results

3.6.1. Conventional Networks. Figure 3.7 shows the speedup provided by our scheduling for over training solely with PM (ratio 0 : 1). The horizontal axis is the ratio of PM to DRAM used to train the network. We observe that when PM is used as a direct substitute for DRAM, performance is poor with a 3x to 8x increase in training time (red horizontal line). However, with a minimal amount of DRAM such as an 8 : 1 PM to DRAM ratio, we are able to dramatically improve performance without changing the overall memory footprint of the application. Further, adding more DRAM only marginally increases performance.

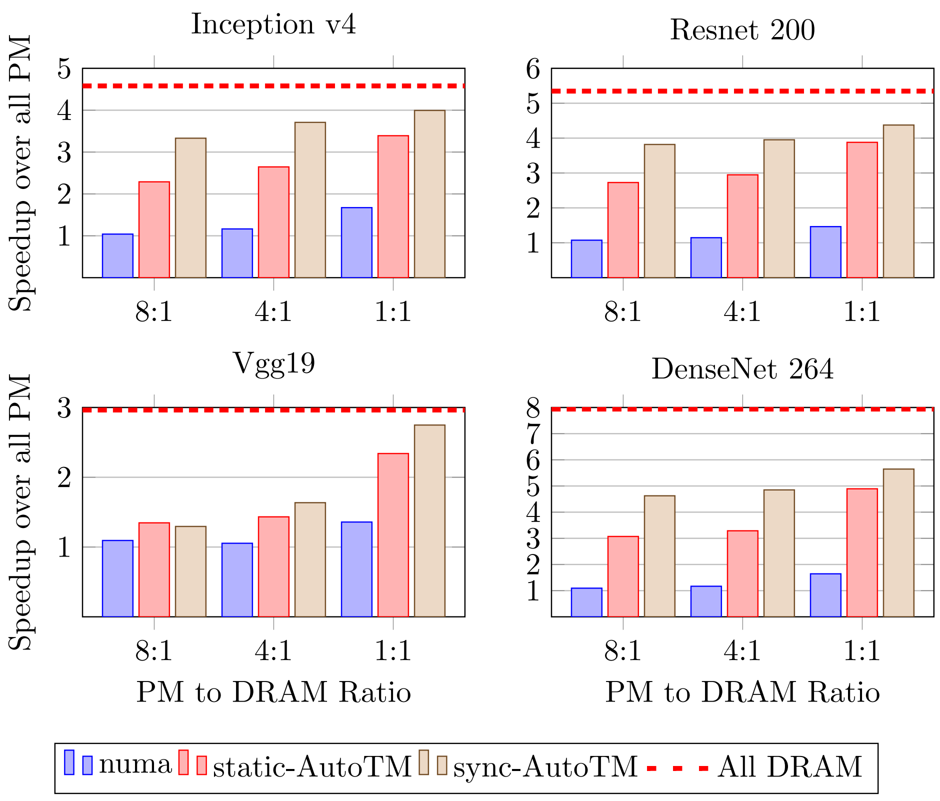


FIGURE 3.7. Results for conventional networks. Note different y-axes. The baseline (1.0 in the graphs) is a system with only PM (ratio of 0 : 1).

This above performance gain does not occur using conventional first-touch NUMA. This is because first-touch NUMA [**56**] works by allocating tensors into DRAM as they are used by the computation graph until the DRAM capacity is reached. In the training of DNNs, tensors produced early on in the forwards pass are used during the backwards pass and thus must be live for the majority of the graph's computation [**89**]. With first-touch NUMA, these long lived tensors are assigned to DRAM forcing future short-lived tensors into PM.

AutoTM, on the other hand, is aware of the performance implication of these long lived tensors. The general strategy AutoTM takes is to prioritize short-lived tensors for DRAM placement (Section 3.6.4). These short-lived tensors mainly include intermediate tensors generated during the backwards pass. By prioritizing short-lived tensors, AutoTM ensures that more tensors overall may reside in DRAM.

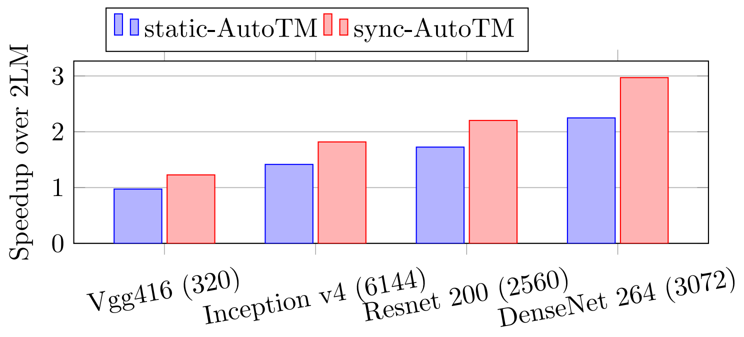


FIGURE 3.8. Performance of the static and synchronous formulations relative to 2LM cached mode.

Vgg is an outlier due to its extremely large second convolution layer. With small DRAM sizes, some or all of the input and output tensors of this large layer must be placed in PM, incurring a performance penalty. Once these tensors can be placed in DRAM, we see a significant performance improvement as can be seen in the performance jump from the 4 : 1 ratio to the 1 : 1 ratio. Another interesting feature of this network is that the *synchronous* formulation performs slightly worse than the *static* formulation for an 8 : 1 ratio. This is caused by the interaction between the insertion of move nodes and the defragmentation procedure.

3.6.2. Comparison to a hardware DRAM cache. We use very large networks to compare AutoTM to a hardware-controlled DRAM cache (2LM mode). The results from the large benchmarks Vgg416 and the large batchsize Inception v4 are shown in Figure 3.8. The *static* formulation has performance comparable to 2LM with the *synchronous* formulation running 23% faster. We see further improvement for the other networks, with Resnet 200 running over 2x faster than 2LM. Inception v4, on the other hand, runs almost 2x faster under the synchronous formulation than under 2LM.

As predicted in Chapter 2, there are several reasons why AutoTM outperforms the DRAM cache. First, AutoTM is aware of the difference between semantically *live* data versus *dead* data and thus elide the unnecessary dirty write-backs on the backward. This can be seen in Figure 3.9, which shows the trace of bandwidth through out a single iteration of training for the large DenseNet model under AutoTM. Contrast this with Figure 2.5c, showing the 2LM bandwidth for the same model. AutoTM only generates PM writes during the forward pass (where it is storing intermediate

Table 3.2. Comparison of data moved and execution time for three common CNNs running in 2LM and under AutoTM. All DRAM and PM values are in GB.

	2LM					AutoTM				
	DRAM Read	DRAM Write	PM Read	PM Write	Runtime (s)	DRAM Read	DRAM Write	PM Read	PM Write	Runtime (s)
Inception v4	8338	4254	1019	919	572	8103	3459	543	473	304
Resnet 200	8565	3914	950	903	514	8565	3316	652	467	229
DenseNet 264	7418	3559	1027	969	524	7419	2947	639	510	169

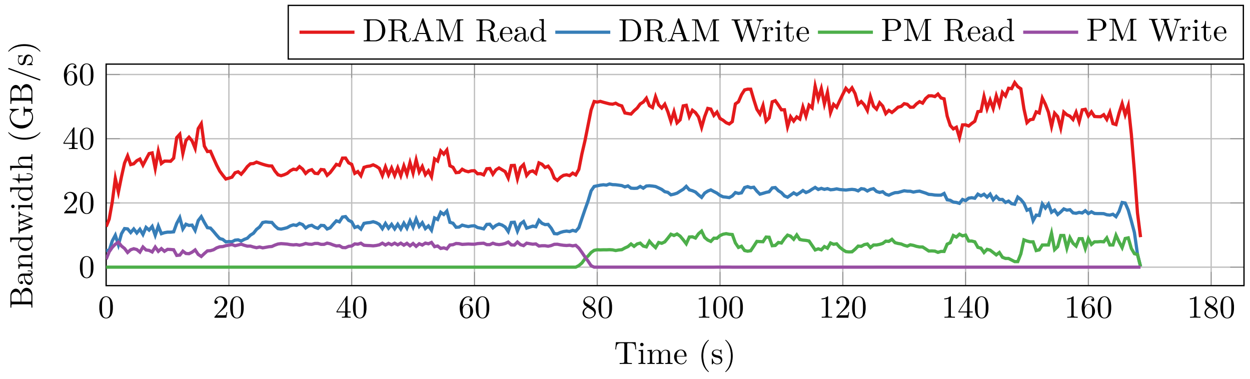


Figure 3.9. Memory bandwidth under AutoTM. Samples are averaged over a 2.5 second sliding window to filter high frequency components.

activations for use on the backward pass). Similarly, AutoTM only generates PM reads during the backward pass. Table 3.2 compares the total amount of data moved for these workloads in 2LM and under AutoTM. AutoTM generates similar amounts of DRAM traffic, but only 50% to 60% of the PM traffic.

The average read and write bandwidth that AutoTM achieves is to PM is also significantly higher than that achieved during 2LM. This is because AutoTM is designed to read and write to PM in the patterns discussed in Section 2.3 for achieving high bandwidth. However, the average bandwidth in Figure 3.9 does not tell the whole story. Under AutoTM, tensors are usually moved between DRAM and PM (and vice versa) synchronously between compute kernel execution. Therefore, during kernel execution, there is no data movement. Thus, we are seeing the bandwidth averaged over times of data movement and times of no data movement, implying the active bandwidth is much higher.

3.6.3. Cost-Performance Analysis. Does PM deliver a cost performance advantage over DRAM for training large DNNs? Table 3.3 provides a summary of module cost and cost per GB

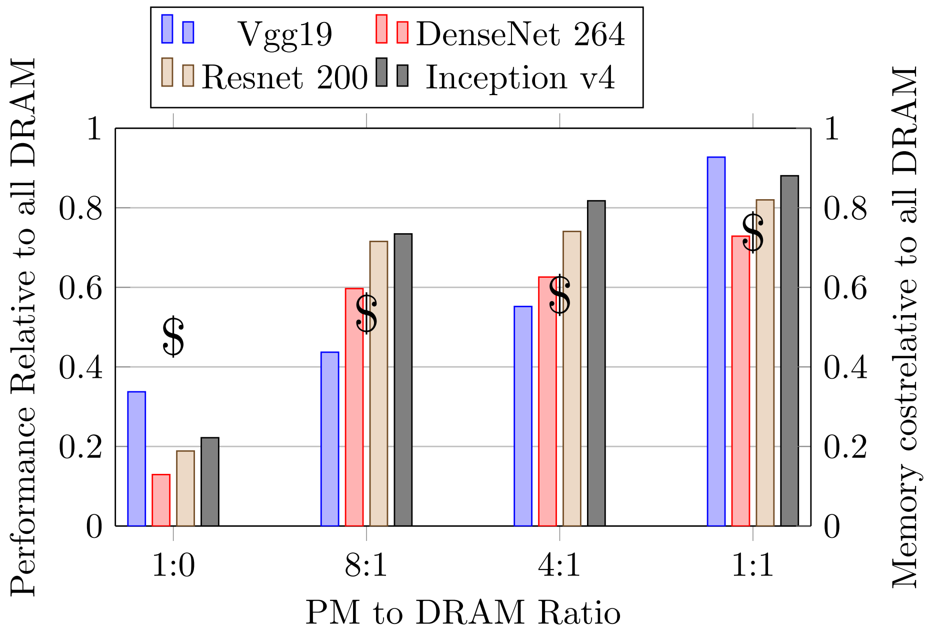


FIGURE 3.10. Price–performance analysis. The bars (left axis) show the network performance relative to all DRAM while the dollar signs (right axis) show the memory system price relative to all DRAM. The regions where the bars are higher than the dollar signs are regions where price–performance is lower.

for a selection of server class DRAM and Optane DIMMs, as quoted by Lenovo[2]. The price per GB of DRAM stays roughly constant across module sizes. PM, on the other hand, increases in price per GB as capacity increases. Prices are driven by business decisions. Because a 512 GB DRAM DIMM is not available, a premium can be charged for this capacity module.

For our analysis, we use the price of the cheapest PM at \$7.85 per GB and the cheapest DRAM at \$16.61 per GB. This means the cost-per-GB advantage of PM over DRAM is about 2.1x. In Figure 3.10 we only include the cost of the memory actually used. Since Optane DC is a new technology, prices are still adjusting, and as the technology matures, price will likely decrease, improving its cost-effectiveness.

Figure 3.10 shows the relative performance of AutoTM for our workloads (bars, left axis) as well as the cost of memory used by the application relative to the case where all DRAM is used (dollars, right axis). The use of PM can be cost effective if the performace lost by replacing some DRAM with PM is less than the cost reduction. We observe that only using PM directly is not

[2]accessed August 14, 2019

Capacity (GB)	Price per DIMM	Price per GB
DRAM 8	$190.45	$23.81
DRAM 16	$265.82	$16.61
DRAM 32	$602.50	$18.83
DRAM 64	$1,255.75	$19.62
DRAM 128	$2,512.00	$19.63
Optane 128	$1,004.50	$7.85
Optane 256	$3,466.75	$13.54
Optane 512	$10,552.00	$20.61

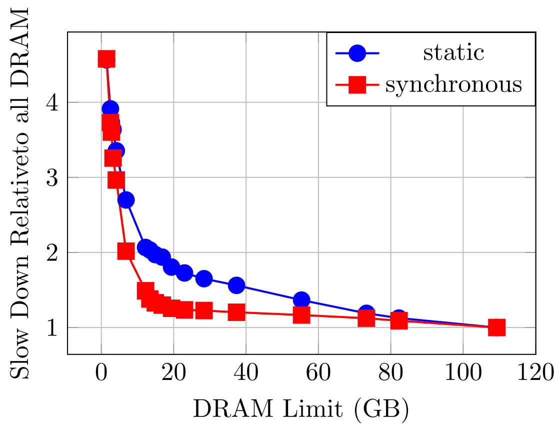
TABLE 3.3. Lenovo price summary of Optane and server class DRAM. (see footnote 2)

cost effective, the performance loss caused by the slower devices is not offset by the lower price. However, for PM to DRAM ratios of 4 : 1 and 1 : 1, AutoTM can provide a cost-performance benefit. This cost-performance benefit may be reduced when taking the whole system into account, but the cost of memory is usually the dominant cost in large systems.

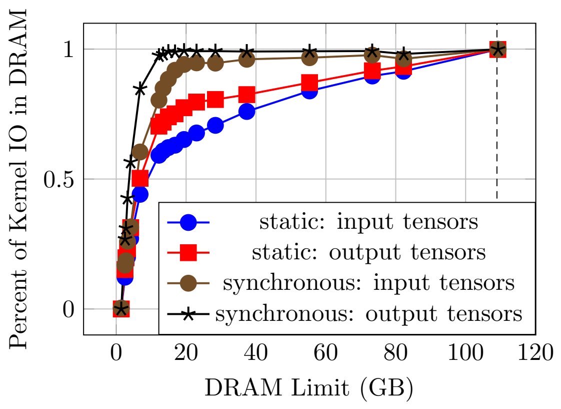
3.6.4. Understanding the ILP Solution. In this section, we present some insight to how and why AutoTM works using Figure 3.11. Figure 3.11a shows the slowdown of the *static* and *synchronous* relative to all DRAM. With a small amount of DRAM, performance improves rapidly. This trend continues until a critical threshold where adding DRAM yields diminishing returns.

To understand this behavior, we look at the input and output memory locations for each kernel as well as the amount of data moved. Figure 3.11b shows the percent by memory footprint of kernel input and output tensors in DRAM. We see a trend to assign as many kernel inputs and outputs into DRAM, with a slight priority on output tensors. This is consistent with the lower write bandwidth of PM. Furthermore, the point where almost 100% of output/input tensors are in DRAM corresponds to the critical point in the performance graphs. This implies a general strategy to maximize kernel read and write memory accesses in DRAM, followed by data movement to PM when DRAM capacity constrained.

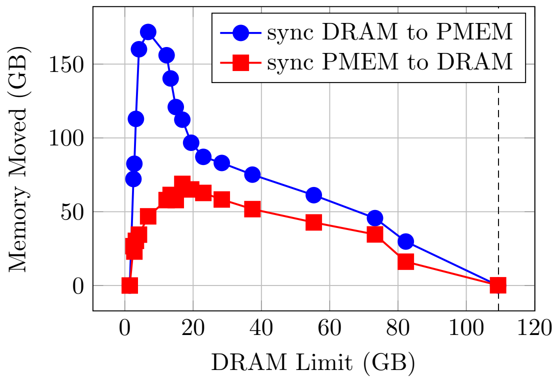
This idea is reinforced by Figure 3.11c, which shows the total amount of memory moved between DRAM and PM in the *synchronous* formulation. With a DRAM limit near zero, no data movement occurs since no data may be moved into DRAM. A small DRAM allowance, however, is followed by a dramatic increase in data movement, again with an emphasis on moving data from DRAM to PM. Once the DRAM limit allows almost all tensor inputs/outputs to reside in DRAM, the



(a) Slowdown relative to all DRAM.



(b) Percent by memory size of all kernel inputs and outputs in DRAM.



(c) Amount of data moved between DRAM and PM.

FIGURE 3.11. AutoTM's solution strategy for Inception v4.

amount of data movement decreases. The region of gradual slowdown seen in the performance plot is caused primarily by data movement rather than kernel slowdown from more memory accesses to PM.

3.6.5. Kernel Profiling Accuracy. To evaluate the accuracy of our profile based approach, we show the error between the expected runtime and the measured runtime in Figure 3.12. The worst case error occurs for in the *static* formulations for DenseNet 264 (19%). This error is likely due to CPU caching. During profiling, move nodes are placed at the inputs of kernels under test to allow the inputs and outputs of the kernel to be varied between DRAM and PM. Kernels cannot be directly profiled due to levels of indirection used in nGraph. Because move nodes are implemented

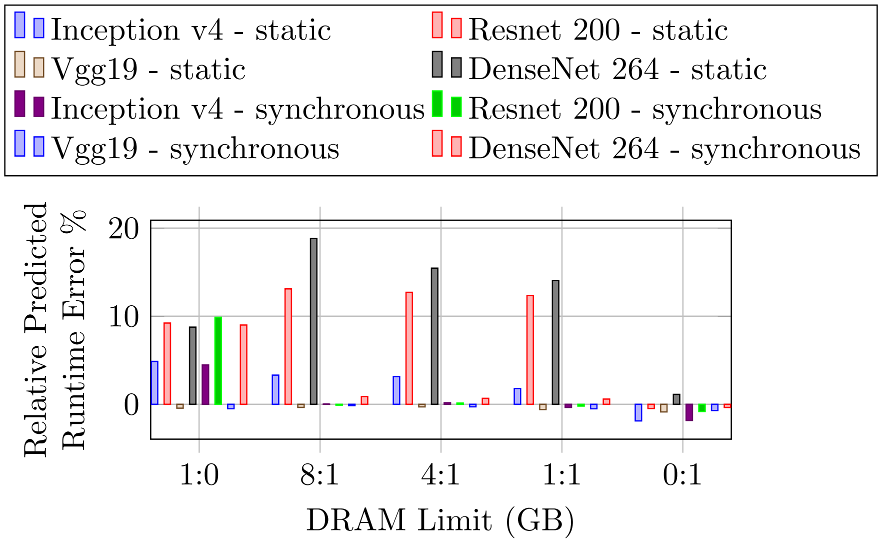


FIGURE 3.12. Comparison of actual execution time and execution time predicted by kernel-wise profiling for the conventional networks.

using streaming instructions, no data is resident in CPU caches following these instructions. Hence, our profiling step is essentially measuring the cold-performance of these kernels. This results in an *overestimation* in run time for the *static* formulation since no move nodes are used. Vgg19 is less affected due to its very large intermediate layers.

The expected runtime for the *synchronous* formulation closely follow the predicted runtime because of the use of move nodes placed in the computation graph. The error in the 1 : 0 all PM case exists for similar reasons.

3.6.6. ILP Solution Times. It is important that the memory optimizer is able to run in a reasonable amount of time. Although ILP is inherently NP-hard, recent solvers can find solutions to many problems quickly. Table 3.4 shows the total amount of time optimizing the ILP. The number of retries due to memory fragmentation is shown in parentheses. Solution time increases with model complexity. Since the optimized computation graph will run for days or weeks to fully train the DNN, this optimization overhead will be amortized. The worst case is the static formulation for DenseNet which takes a little less than an hour to fully solve.

	Static			Synchronous		
Network	8 : 1	4 : 1	1 : 1	8 : 1	4 : 1	1 : 1
Vgg19	0.40 (1)	0.70 (2)	0.82 (2)	2.5 (5)	1.7 (3)	0.94 (2)
Inception v4	37.9 (5)	16.4 (2)	13.7 (2)	50.3 (6)	15.3 (2)	16.4 (2)
Resnet 200	2846 (2)	3105 (3)	91.9 (1)	710 (2)	571 (2)	79.9 (2)
DenseNet	3307 (1)	2727 (1)	2582 (1)	1448 (3)	2021 (3)	1404 (2)

TABLE 3.4. Gurobi ILP solver time to a relative MIP gap of 0.01 for the *static* and *synchronous* formulations for the conventional networks. Entries of the form a (b) indicate the total time a in **seconds** it took to solve the ILP b times. Multiple solutions are needed in the case of memory fragmentation management.

3.7. Extending AutoTM

In this section, we discuss two extensions to AutoTM: allowing *asynchronous* data movement and performing kernel implementation selection. We explain why these extensions were not included in the original formulation and demonstrate their viability on a CPU-GPU platform. These extensions and the GPU implementation of AutoTM show that it is a general and flexible framework for managing heterogeneous memory.

The first extension we investigate is *asynchronous* offloading and prefetching of intermediate tensors between memory pools. This allows data movement to be overlapped with computation, improving the throughput of the application as a whole. We implemented asynchronous data movement on the PM system, but found it performed poorly on existing CPU only systems for a number of reasons. Neither a dedicated copy thread nor DMA provided sufficient performance to mitigate the overhead of these approaches. However, a PCIe connected GPU offers a high speed asynchronous data copy API, which is ideal for implementing this extension.

The second extension to the formulation is performing kernel implementation selection. The underlying library used by nGraph to perform forward and backward convolutions for the GPU backend is cuDNN [**16**], a deep learning library from Nvidia. This library exposes several different implementations for each convolution, each with performance and memory footprint tradeoffs. Generally, faster implementations require more memory. In a memory starved case, this larger memory footprint may require more offloading of previous tensors, resulting in a global slowdown.

Since nGraph does not expose any kernel selection options for the CPU backend, we implement this on the GPU instead.

3.7.1. ILP Formulation Modifications. Since AutoTM is implemented using an ILP formulation, we can extend it to be aware of the performance and memory footprint of these different kernels and globally optimize tensor movement and implementation selection. Here, we provide a high level overview of the additions to the ILP formulation to express asynchronous data movement and kernel implementation selection.

3.7.1.1. *Objective Function:* In our formulation, we allow an arbitrary number of tensors to be moved between GPU and CPU DRAM concurrently with a single kernel. This results in a new objectives function

$$\min \sum_{k \in \mathcal{K}} \max \left\{ \rho_k, \sum_{t \in \text{ASYNC}(k)} M_{t,k}^{\text{async}} \right\} + \sum_{t \in \mathcal{T}} M_t^{\text{sync}} \tag{3.7}$$

where $\text{ASYNC}(k) = \{t \in \mathcal{T} : t \text{ can be move concurrently with } k\}$ and $M_{t,k}^{\text{async}}$ is the amount of time (if any) spent moving tensor t during the execution of k. The max operation is implemented using standard ILP techniques.

3.7.1.2. *Tensor Graphs:* We must extend the tensor flow graphs $\mathcal{G}_t$ to encode points of asynchronous tensor movement. We identify kernels that can be overlapped with data movement and add a component in each tensor's graph (like those shown in Figure 3.6b) for each kernel with which the tensor can be moved concurrently.

3.7.1.3. *Asynchronous Data Movement:* Asynchronous move times for tensor t must be generated for each kernel k across which t may be moved. This comes directly from the extended tensor graph

$$M_{t,k}^{\text{async}} = \left(\frac{|t|}{\text{BW}_{P \to D}^{\text{ASYNC}}} \right) e_{P \to D} + \left(\frac{|t|}{\text{BW}_{D \to P}^{\text{ASYNC}}} \right) e_{D \to P} \tag{3.8}$$

where $e_{P \to D}$ ($e_{D \to P}$) is the binary edge variable in $\mathcal{E}_t$ corresponding to the asynchronous movement of t from PM to DRAM (DRAM to PM) across kernel k.

3.7.1.4. *Selecting Kernel Implementations:* Let $\mathcal{I}(k) = \{1, 2, \ldots, n_k\}$ be an enumeration of the implementations for kernel k. We generate one-hot binary variables $v_{i,k}$ for all $i \in \mathcal{I}(k)$ where $v_{i,k} = 1$ implies implementation i is to be used for kernel k.

3.7.1.5. *DRAM Constraints:* Constraining DRAM is similar to the *static* and *synchronous* formulations, but now includes kernel memory footprints with

$$\sum_{i \in \mathcal{I}(k)} s_{k,i} v_{k,i} + \sum_{t \in \mathbb{IO}(k)} t^{\texttt{DRAM}}_{t,k} + \sum_{t \in \mathbb{L}(k)} t^{\texttt{DRAM}}_{t,\mathrm{ref}(k)+} \leq \mathcal{L}_{\mathrm{DRAM}} \tag{3.9}$$

where $s_{k,i}$ is the memory footprint of implementation i of k.

3.7.1.6. *Kernel Timing:* The expected runtime of a kernel is now dependent on which implementation of the kernel is chosen. Building on the example given in Section 3.3.5, assume that k has two implementations (i.e. $\mathcal{I}(k) = \{1, 2\}$). The expected execution time for ρ_k kernel k is modeled as

$$\rho_k = \sum_{c \in \mathcal{C}(k)} \sum_{i \in \mathcal{I}(k)} n_{k,c,i} (d_{k,c} \wedge v_{i,k}) \tag{3.10}$$

with $n_{k,c,i}$ is the profiled runtime of implementation i of kernel k in IO configuration c. This approach does not account for the performance impact of memory conflict between data movement and the computation kernel. However, the maximum memory bandwidth of our GPU is 616 GB/s while the maximum bandwidth of PCIe is 16 GB/s. Thus, the impact of asynchronous data movement is likely low.

3.7.2. Implementation. We modified the GPU backend of nGraph to support synchronous and asynchronous tensor movement as well as to allow for kernel selection of forward and backward convolution kernels. All GPU kernels are profiled with inputs and outputs in GPU memory. When implementation selection is available, all possible implementations of a kernel are profiled as well. Asynchronous movement was implemented using two CUDA [**69**] streams: one for computation and the other for data movement via *cudaMemcpyAsync*. These streams are synchronized before and after an asynchronous movement/computation overlap to ensure data integrity.

3.7.3. Methodology. Our system used a Nvidia RTX 2080 Ti with 11 GB of GDDR6 using CUDA 10.1 and cuDNN 7.6. The host system was an Intel Core i9-9900X with 64 GB of DDR4 DRAM.

We use the same convolutional neural networks used earlier. The networks and batch sizes used are given in Table 3.1. We compare the results of AutoTM with the performance of *cudaMallocManaged*, which is a memory virtualization layer offered by Nvidia for automatically moving data from the CPU to the GPU in the event of a GPU page fault and moving unused pages from GPU DRAM to CPU DRAM.

3.7.4. GPU Results. The results for the GPU experiments are given in Figure 3.13. For networks that fit on the GPU, our approach has no overhead as the ILP optimizer realizes no data movement is needed. As the intermediate working set increases, we observe a several fold improvement with AutoTM over *cudaMallocManaged* due to the lack of runtime overhead of our approach and its algorithm awareness. AutoTM provides considerable speedup when data movement between the CPU and GPU is required. The *asynchronous* extension outperforms the *synchronous* formulation with its ability to overlap data movement and computation. However, the *asynchronous* extension is limited to overlapping tensor movement with a single kernel at a time. Since the RTX 2080 Ti executes kernels faster than data movement, time must be spent to synchronize the two CUDA streams.

The synchronization overhead of overlapping tensor movement with a single kernel can be seen by comparing the achieved performance with the theoretical best performance, calculated by assuming infinite GPU DRAM capacity and using the fastest possible implementations for all kernels. As the memory requirement for training increases, AutoTM achieves a lower fraction of this best performance due to synchronization.

We did not compare our results directly against vDNN [**89**] for two reasons. First, the RTX 2080 Ti GPU is much faster than the Titan X used in that work and thus we cannot compare results directly. Second, the code for vDNN is not available, making direct testing on our GPU difficult. However, while vDNN leverages the same characteristics as AutoTM (communication overlapping, kernel selection, and liveness analysis), AutoTM uses mathematical optimization rather than heuristics providing a more general solution.

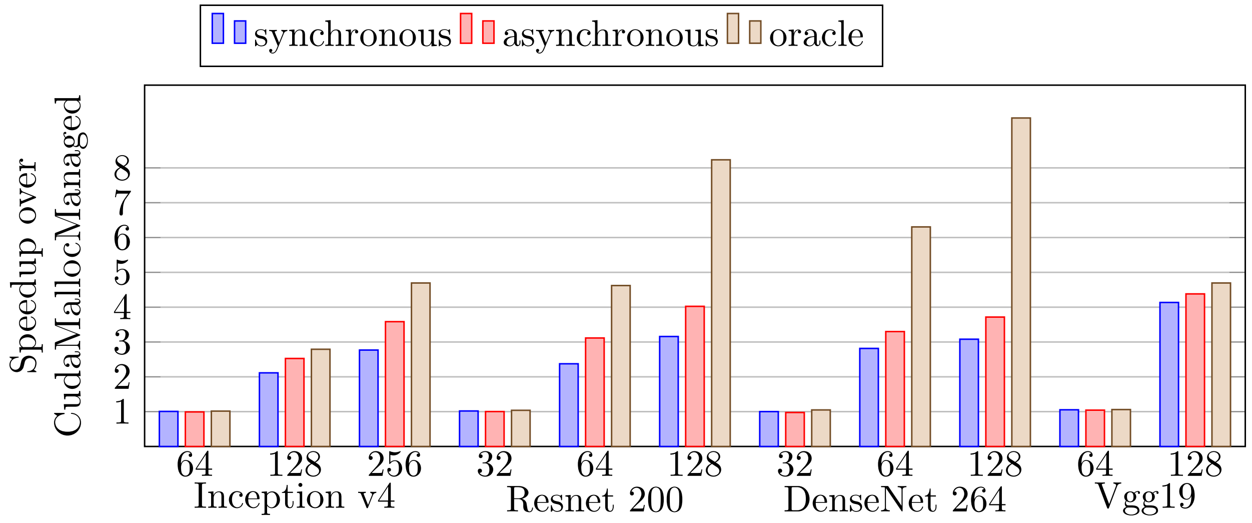


FIGURE 3.13. GPU performance of AutoTM relative to *cudaMallocManaged*.

3.8. Related Work

As an emerging technology Intel Optane DC has been explored in several recent works. These include in depth performance analysis [**48**], large graph analytics [**33**], and database I/O primitives [**103**]. Research into using Optane PM for virtual machines demonstrates that only a small amount of DRAM is needed [**45**]. Flash based SSDs have also been used to reduce the DRAM footprint in database [**30**] and ML [**31**] workloads. These approaches use a software managed DRAM cache to mitigate the slow performance and block level read/write granularity of NVM SSDs. Operating system support for managing heterogeneous memory [**2**, **108**] and support for transparent unified memory between GPU and CPU [**49**, **74**] have been studied extensively in the past. However, to the best of our knowledge, the proposed work is the first to explore the design space and cost–performance tradeoffs of large scale DNN training on systems with DRAM and PM.

Previous works such as vDNN [**89**] exploit heterogeneous memory between GPUs and CPUs by recognizing that the structure of DNN training computation graphs has a pattern where intermediate tensors produced by early layers are not consumed until much later in the graph execution. The authors of vDNN exploit this to develop heuristics for moving these tensors between GPU and CPU DRAM during training to free GPU memory. SuperNeurons [**106**] and moDNN [**15**] build on vDNN. SuperNeurons introduces a runtime manager for offloading and prefetching tensors between

GPU and CPU memory as well as a cost-aware method of applying recomputation of forward pass layers during the backward pass to reduce memory. Similar to our approach, moDNN allows tensors to be offloaded and uses profiling information of kernel runtime and expected transfer time to determine how it will overlap computation and communication. AutoTM differs from these previous approaches in that we use mathematical optimization rather than problem specific heuristics. AutoTM also generalizes the location of data across DRAM and PM instead of requiring data to be in DRAM for computation.

Integer Linear Programming and profile guided optimization have been used widely to address similar problems in research literature. For example, work in the embedded system space [**4**] uses ILP in to optimize the allocation of heap and stack data between fast SRAM and slow DRAM. ILP has also been used in register allocation [**36**] and automatic program parallelization [**40**]. ILP has been used to optimize instruction set customization and spatial architecture scheduling [**70**]. Profile guided optimization has been used for dynamic binary parallelization [**112**], process placement on SMP clusters [**14**] and online autotuning of CPU and GPU algorithm selection [**78**]. AutoTM builds on these ideas to address the new problem of data movement in heterogeneous memory systems.

3.9. Conclusions

We present AutoTM, an ILP formulation for modeling and optimizing data location and movement in static computation graphs such as those used for training and inference of DNNs. AutoTM uses profile data to optimally assign kernel inputs and outputs into different memory pools and schedule data movement between the two pools to minimize execution time under a memory constraint. With AutoTM, we can obtain 2x performance improvement over hardware DRAM caching solutions. We further find Intel Optane PM can reduce the DRAM footprint of DNN training by 50 to 80% without significant loss in performance. Given the lower cost of Optane PM, this can yield a cost-performance benefit in systems with mixed DRAM and PM over a system with only DRAM.

AutoTM uses minimal problem specific heuristics, making it generally applicable to different systems and networks. We demonstrate this flexibility by extending AutoTM to GPUs, and believe it can be further extended to further heterogeneous systems, such as those with multiple GPUs or multi-level systems with HBM, DRAM, and PM.

CHAPTER 4

Generalizing Heterogeneous Memory Management

4.1. Introduction

In the previous chapter, we presented AutoTM [**44**], an ILP based technique for managing tensor location and movement during CNN training. When implementing this in the ngraph compiler, the compiler and runtime itself was modified in order to use multiple levels of memory. That is, we *added* bespoke heterogeneous memory support to an existing framework. This can lead to scalability issues - for every framework, we must add support for data tiering. Furthermore, as more memory technologies come available, support for them in each modified framework must be added individually at the cost of developer effort.

In this chapter, we will explore ideas to generalize of the heterogeneous memory management techniques demonstrated in AutoTM. We explore a possible memory management/data-tiering runtime upon which multiple applications can be built. *CachedArrays* is a new memory management framework which allows programmers (or the runtime) to direct the heterogeneous memory aware memory manager via application-specific policies. Table 4.1 shows how *CachedArrays* compares to other data management (data tiering) solutions.

In this work, we focus on a unique layer in the stack: the compiler and the runtime of a managed language. By focusing on this layer, we can gain the benefits of algorithmic-specific optimizations

Work	**Abstraction Layer**	**Granularity**	**Programmability**	**Mechanism**
SAGE [**26**]	Algorithm	Data Structure	Application Specific	Maunally Partitioned Data Structures
AutoTM [**44**], Sentinel [**87**]	Compiler	Tensor	Transparent	Profile Guided Optimization
vDNN [**89**]	Application	Tensor	Application Specific	Manual Partitioning
Nimble [**108**], KLOC [**54**], Thermostat [**2**]	Operating System	Page	Transparent	Virtual Memory
Memory Mode (in Intel PMem))	Hardware	Cache Block	Transparent	HW Managed Cache
Kona [**11**]	Hardware	Cache Block	Transparent	SW Runtime based on Cache Coherence
CachedArrays (This work)	Application	Variable Sized Object	Transparent (optional annotations)	Type System/Runtime

TABLE 4.1. Landscape of Related Work in Data Tiering in Heterogeneous Memory Systems.

through simple programmer annotations and track data status at the *object* granularity. We do not track metadata at a fixed fine (64-byte block or page) granularity, and we are not restricted only to tensors or specific data structures. Additionally, by targeting the compiler/runtime layer, our techniques can improve performance when combined with other data management techniques (e.g., DRAM caches).

We present a case study focusing on training large CNNs since this workload has a regular and easy to exploit data reuse pattern. We show that our generalized framework can provide similar performance improvements as the specialized AutoTM implementation [**44**]. In Chapter 5.1, we will further generalize *CachedArrays* to apply these ideas to a DLRM workload that has significantly different usage patterns than the kernel-based CNNs studied so far.

In this chapter:

- We describe *CachedArrays*, an approach to heterogeneous memory management which separates the data management policy from the data management engine and bridges the semantic gap between the algorithm-level *objects* and the underlying memory *devices*.
- We implement *CachedArrays* in Julia and show how programmers can either transparently use *CachedArrays* or provide hints for more efficient data management.
- We present a case study using *CachedArrays* for training deep learning models and demonstrate *CachedArrays* provides 1.19 to 1.74× speedup over a real hardware DRAM cache for CNN training.

Rest of the chapter is organized as follows. In Section 4.2 we present high level overview of our approach with examples. In Section 4.4 we describe the detailed implementation of the proposed framework in Julia. This will be followed by a detailed case study in Section 4.7 with results from experiments on real hardware. We end the chapter with a discussion in Section 4.9.

4.2. A Generic Heterogeneous Memory Management System

From prior work in this area (both that highlighted in Table 4.1 as well as that conducted in the previous chapters), we can make two observations:

(1) There will not likely be a single one-size fits all implementation for heterogeneous memory management. For example, mechanisms suitable for large data sizes will likely fail for workloads that require fine-grained management.

(2) Even with a strong data movement implementation, the policy that decides where to place data and when to move will vary from application to application, requiring program-level semantic information.

To that end, we explore the *common* characteristics between all heterogeneous memory management systems. We develop a more disciplined approach to heterogeneous memory management by implementing a generic system that can be specialized to suit a particular application. Figure 4.1 presents a high level overview of a generic heterogeneous memory management system. The programmer/application/runtime (summarized as the *abstract runtime*) interacts with *objects*, each of which is associated with one or more *regions.* Regions are the unit of memory management and themselves reside on *devices* (e.g., DRAM or PM). The *system* (right side of the figure) consists of all the devices capable of hosting regions and some amount *state* that helps track the relationships between regions and objects.

Between the abstract runtime and the system sits the *policy*, acting as a bridge between the two. The primary goal of the policy is to coordinate the assignment of *objects* and *regions* to improve the performance of the abstract runtime. This can involve making intelligent decisions about when *objects* are assigned *regions* in a faster memory and when to migrate the primary region for an object from a faster memory to a slower memory. To facilitate this, the policy uses the data management API exposed by the system. The system may also be able to help in the decision process by returning runtime statistics about *objects* and *regions*, such as the number of read and write requests. The abstract runtime influences the location and properties of its objects through an API exposed by the policy. Table 4.2 provides common terms and definitions associated with this generic approach.

4.2.1. *Memory Mode* as an Instance of a Generic Heterogeneous System. To make the discussion more concrete, we will explore how the memory controller in Intel's *memory mode* (2LM) follows the pattern of this generic heterogeneous memory management framework and implements the ideas outlined in Table 4.2. First, we identify the main components.

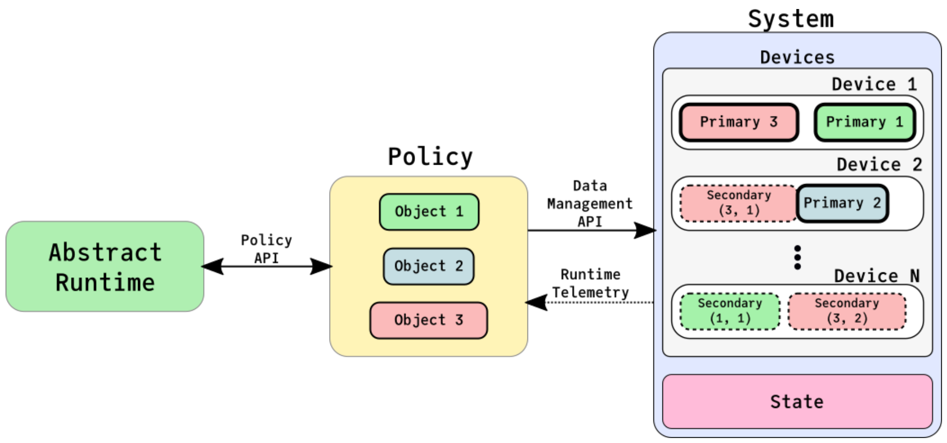


FIGURE 4.1. High level idea of a generic heterogeneous memory management system. The *abstract runtime* interacts with *objects* which are backed by *regions* located on some device. The *abstract runtime* influences the location of *objects* using the policy API. The *policy* communicates with the *system* through a data management API. The *system* consists of one or more devices, each capable of hosting multiple *regions*, and state which maintains the relationship between objects and their regions/primary region. Each device in the manager can have different properties (e.g., speed, persistence). For best performance, the policy implementation should consider the constraints imposed devices.

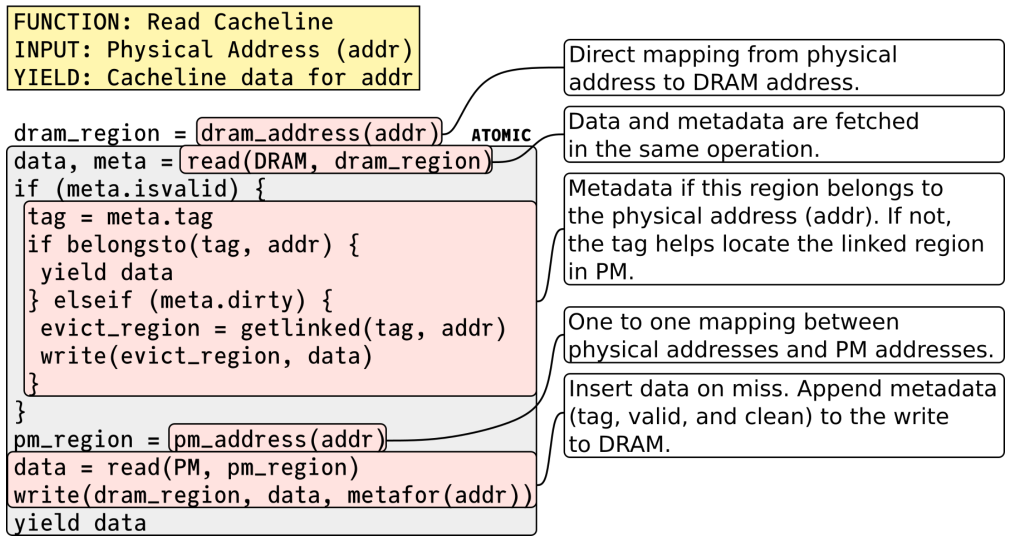
- **Devices**: The 2LM memory controller consists of two devices, DRAM and persistent memory (PM). While physically these are made up of several components (i.e., the DIMMs whose address space is interleaved), we will treat these as a unified device with a contiguous physical address space.
- **System**: The system consists of the devices mentioned above and metadata. The metadata is composed of several items. First, since the DRAM cache in *memory mode* is direct mapped, there is an implicit mapping from physical cache line address to potential DRAM cache line address (i.e., DRAM *region*) using the modulo of the total DRAM size. Second, each DRAM cache line has associated metadata stored in previously unused ECC bits. This metadata includes a valid bit, a physical tag of the upper-order bits of the region's linked PM region (cache line address), and a dirty bit indicating if this cache line must be written back to PM upon eviction.
- **Objects**: In this system, an object is a physical cache line address. From the CPU's perspective, it simply asks the memory controller for the data corresponding to a cache

Term	**Definition**
System	A system consists of one or more *devices* and state. The state can include the assignment of *objects* to *regions*, relationships between *regions*, policy specific state, etc.
Policy	Intermediate layer between the system and abstract runtime. The goal of the policy is coordinate the assignment of *objects* to *regions* in order to improve performance of the abstract runtime.
Abstract Runtime	The entity that interacts with the policy that actually uses and manipulates *objects*. This can be an application, a runtime environment, the OS etc.
Devices	Sources of memory like DRAM or memory heaps. Devices host the memory for *regions* and are spanned (either implicitly or explicitly) by the regions contained within.
Object	An object is an application entity for memory management. For example, in machine learning workloads, an *object* might be a "tensor" (a multidimensional strided array). For hardware caches, an *object* might be a physical address (or more precisely, the base physical address of a cache line).
Region	A unit of contiguous memory, used as the backing store for a "objects".
Linked Regions	Two regions are *linked* if they both belong to the same object. For example, in a DRAM cache, a valid cache line in DRAM is linked to its parent cache line in PM.
Primary Region	When two or more regions belong to the same object - there is the question of what actions to take when writes are performed to the object. Are all regions updated, or is a single region updated and marked as dirty with respect to the other regions? In the case of the latter, the region that contains the most up-to-date version of the data is the *primary* region.

TABLE 4.2. Common patterns and phenomena in a generic heterogeneous memory management system.

line address. The memory controller is the entity responsible for mapping this address to the DRAM or PM devices and retrieving the data.

- **Regions**: Regions are references to the actual backing memory behind objects. In the case of the 2LM memory controller, this is a *device* address in either DRAM or PM. The memory controller uses the state described above to determine and update which *regions* (device addresses) are associated with *objects* (physical address).
- **Policy**: In the case of the memory controller, the policy is built into the hardware as a state machine. To the CPU and LLC, the policy exposed by the memory controller is a



```
FUNCTION: Read Cacheline
INPUT: Physical Address (addr)
YIELD: Cacheline data for addr

dram_region = dram_address(addr)
                                         ATOMIC
data, meta = read(DRAM, dram_region)
if (meta.isvalid) {
 tag = meta.tag
 if belongsto(tag, addr) {
  yield data
 } elseif (meta.dirty) {
  evict_region = getlinked(tag, addr)
  write(evict_region, data)
 }
}
pm_region = pm_address(addr)
data = read(PM, pm_region)
write(dram_region, data, metafor(addr))
yield data
```

Direct mapping from physical address to DRAM address.

Data and metadata are fetched in the same operation.

Metadata if this region belongs to the physical address (addr). If not, the tag helps locate the linked region in PM.

One to one mapping between physical addresses and PM addresses.

Insert data on miss. Append metadata (tag, valid, and clean) to the write to DRAM.

FIGURE 4.2. Pseudo-code algorithm for an "insert-on-miss" read operation for the memory controller when in *memory mode*. In this system, DRAM regions (i.e., cache line) are linked with PM regions using a tag. If an object (physical address) has a DRAM region, than that region is implicitly its primary region.

very simple API: `read_cacheline` and `write_cacheline`. All steps taken by the policy happen as side effects of these two operations.

- **Abstract Runtime**: The *abstract runtime* is the user of the API exposed by policy. Thus, the runtime is the CPU system as a whole (anything capable of emitting read or write requests to the memory controller).

Figure 4.2 outlines pseudo-code modeling the memory controller's insert-on-miss cache line read. Recall, the *policy* uses functionality exposed by the *system* to update state and initiate reads and writes to the underlying devices. In particular, the system supports logical functions like `belongsto` which queries whether a DRAM region belongs to a particular physical address (implemented by comparing the tag with the appropriate bits of the address) and discovering linked regions (again making use of the tag).

One important thing to note is that while the policy logically exists separate from the system (implemented by synthesizing behavior of the system), the policy is still dependent on and limited by the system. For example, the DRAM cache implemented by Intel's *memory mode* is direct mapped because the system cannot efficiently maintain the metadata required for a higher-associativity cache.

4.2.2. Separation of Objects and Regions. At first glance, the separation of the idea of an *object* (the abstract entity through with the runtime interacts) and *region* (the contiguous memory backing the object) may seem a little strange. However, there is significant precedent for this separation.

For example, consider the vector implementation from the C++ standard template library (STL). The vector consists of a start pointer, an end pointer, and a pointer to the maximum capacity to which the vector can grow with its current allocation.[1] This demonstrates a separation of an object (the vector itself) from the region (virtual memory) backing the object. When the user calls a function like `push_back` on the vector, the vector *may* need to reallocate its backing memory if it runs out of capacity. In other words, the same object is reassigned to a new region *without* the user needing to worry about such low level details.

Another example more closely related to heterogeneous memory comes from virtual memory. Applications running on top of modern operating systems use a virtual address space which the OS and hardware cooperatively assign to physical addresses on a page level granularity. In this context, a virtual page is the *object* and the physical page is the *region*. Existing mechanisms like NUMA migration and Nimble pages [**108**] exploit this by reassociating the virtual address with a new physical address, potentially on a different NUMA node.

4.2.3. Linked Regions. In the case of our previous example of virtual memory, there is generally a one-to-one correspondence between objects (virtual pages) and regions (physical pages).[2] With current virtual memory implementations, it does not necessarily make sense for a virtual page to be assigned to *multiple* physical pages. However, there are many examples in which multiple regions are used to back a single object.

Take, for instance, the CPU cache hierarchy where we can take the view that physical cache line addresses are *objects*. Then regions are either addresses in main memory (if the corresponding cache line is not cached) or some location within the CPU cache system. If a particular line is in the CPU cache, then there are at least two regions backing that physical address: the cached location of the line and its home in main memory. Indeed, there may even be multiple instances

[1]The vector may also contain a reference to its allocator if a custom allocator is used.
[2]Though it is possible for multiple virtual pages to map to the same physical page.

of the cache line all throughout the CPU if multiple cores are reading the same data, all of which are *linked* because they back the same physical address. When a write occurs to a cache line, the cache coherence protocol ensures that all stale copies of that cache line are invalidated. In this case, the location in the CPU cache of that written line becomes the *primary region* as it is the most up-to-date copy of the data. This will eventually be written back to the corresponding region in main memory when that line is evicted from the CPU cache.

Turning back to the virtual memory example, previous works investigating Cache-Only Memory Architectures (COMA) [**32**] would use a node-local page cache for pages located on a remote node. Though COMA style memory management generally did not make its way into mainstream architectures (which instead generally use some form of cache-coherent NUMA), the page cache in COMA provides another example of using multiple *regions* (i.e., physical pages - one local in the page cache, others potentially distributed among other nodes) to back a single object (i.e., data page).

Nonlinked Regions in "Fast Memory" One limitation of the previous example of the CPU cache hierarchy is that regions in "fast memory" (i.e., in the cache) *almost always* have linked regions in main (slow) memory because CPU caches are inclusive with respect to main memory. This does not necessarily have to be the case with general heterogeneous memory systems. In AutoTM, a tensor could live its entire life in DRAM (fast memory) without ever having a region in PM. Conversely, if a tensor ever had a region in PM, that region would persist for the lifetime of the tensor avoiding the need to copy tensors back and forth between the two devices.

4.2.4. Mapping of Framework to Previous Work. Table 4.3 identifies some of the key components of the generic heterogeneous memory management framework in the related work given in Table 4.1. An insight generated by this table is the relative richness versus sparseness of the various interfaces (i.e., the data management API and policy API) and how tightly coupled the components are. For example, the policy in 2LM is very tightly coupled with the system as these are both implemented in hardware in the memory controller. On the other hand, management systems like Kona [**11**] are more flexible as there is a software component that can make high-level decisions about data management that can incorporate better heuristics.

Work	**Entity**	**Description**
vDNN [**89**]	Object	Tensors.
	Region	Virtual memory allocations either in GPU memory or CPU memory.
	System	The vDNN runtime.
	Policy	Built into the runtime, configurable between various static and dynamic strategies.
	Runtime	Applications running in vDNN.
Nimble [**108**]	Object	Virtual memory page.
	Region	Physical memory page either in a near or far memory.
	System	CPU and OS virtual memory subsystem.
	Policy	Integrated policy in the Linux kernel, sparse API.
	Runtime	Applications in userspace.
Memory Mode (2LM)	Object	Physical cacheline address.
	Region	Device Address (DRAM or PM).
	System	Memory Controller.
	Policy	Insert on Miss tightly coupled with memory controller.
	Runtime	CPU Cores (anything generating memory controller read or write requests.
Kona [**11**]	Object	Virtual cacheline address.
	Region	Managed physical address (directed to main memory directly or to a software managed cache for remote memory in main memory).
	System	CPU cache-coherence, page-table management, and runtlime library “KLib”.
	Policy	Baked into “KLib” runtime, coarse API consisting of functions like *malloc* and *free*.
	Runtime	Applications in userspace.

TABLE 4.3. Identification of the various components of the generic heterogeneous memory management framework to existing memory management frameworks.

For interface richness, cache-based schemes like 2LM or Kona do not provide the user with much. In Kona, actions like *malloc* or *free* allow the user to acquire distributed memory that is cached-locally, but don’t allow communication between the user and the policy regarding data semantics. The 2LM DRAM cache has an even sparser user interface as described in Section 4.2.1. Page based mechanisms like Nimble pages [**108**] provides the abstract runtime with more knobs to tune through the Linux kernel’s NUMA subsystem. NUMA allows applications to specifically request allocation of memory on specific nodes, use a generic policy regarding page allocation, or request page migrations. However, without higher-level abstractions, this requires the application to reason about program objects on a page-level basis, which may not always be the most natural approach.

We'd argue that coupling between various components in the heterogeneous memory management framework should be kept as loose as possible and the interfaces as rich as possible. This provides the most modularity for tuning (like making modifications to the policy) and passing program level semantic information to the policy whether through manually inserted annotations or compiler generated hints.

4.3. Basis for the Data Manager and Modular Policy

In this section, we will discuss the driving requirements and APIs behind *CachedArrays*. Recall, the end goal here is a runtime memory management system that can be used both as a building block for AutoTM style machine learning applications as well as future applications where memory management is not as well studied. Julia [**7**] was used as the language to implement this prototype. Julia is a "just-in-time" compiled, garbage collected, dynamically typed language that allows both low-level programs (e.g., memory allocators) and high-level programs (e.g. machine learning training) to be implemented in the same language.

Objects and Regions: Data movement optimization should be done on the "object" level within a program. The object level is where the programmer, compiler, and runtime have specific knowledge of the semantics of the data within their program which can be used to drive data movement and placement considerations. For instance, the programmer knows whether an array will be accessed sparsely or densely, which can affect caching decisions. In DNN-based workloads, memory is passed around as tensors, which are relatively large ($>$ 100s of KiB) contiguous chunks of memory. In these workloads, we would use the tensors, (or more generally multi-dimensional arrays) as the objects to consider for data movement optimization. Other transparent data movement techniques like NUMA and hardware caches lose the semantic information of the application (e.g., data that is semantically dead and will never be re-read may be written back to main memory wasting bandwidth and energy) [**43**].

As with AutoTM, we will focus on machine learning frameworks that use the *kernel programming model*. The kernels may be offloaded to accelerators or optimized CPU primitives (e.g., OneDNN).

Importantly, while *CachedArrays* tracks the use of data on the object level, the objects are backed by array-like constructs. When implemented *above* the language level, this introduces an extra level of indirection as an array reference must traverse from the array to the object, then from the object to the actual backing memory. In kernel programming style, this extra level of indirection has no performance overhead as the extra pointer is dereferenced once and the raw data is passed to the computational kernel.

CachedArrays also supports moving objects between different memory pools and construction of higher order constructs like two-level caches. Thus, *regions*, which are contiguous slices of virtual[3] memory that either hold the current data for an object (which we call the *primary region*) or copies of the data for an object (where we call each copy a *secondary*). Two regions are said to be *linked* if they both are either primary or secondary regions for the same object. These secondary regions may be valid if the primary is *read only*, or *stale* if the primary has been updated and has not propagated these updates to all its secondaries. Currently, *CachedArrays* requires all memory used by an application to be acquired from the OS prior to execution (i.e., no dynamic memory allocation from the OS through system calls like *mmap*). In practice, if an application requires hundreds of gigibytes of memory, then it's likely one of few applications running (to avoid memory contention) and would probably want to minimize system calls for virtual memory anyways because these tend to be quite slow.

4.3.1. Data Manager.

The goal of *CachedArrays* is to separate the mechanism of data tracking and movement from the policy driving this management. Our proposed data manager tracks (1) all *objects* that live in the program, (2) the *primary region* for each objects, and (3) all *linked* regions for all primary primary. The data manager may also contain multiple devices upon which regions may be allocated. In our case studies, these devices are memory heaps for DRAM and PM (persistent memory or NVRAM), but this is not a fundamental restriction of *CachedArrays*.

The data manager supports allocation and deallocation of regions, linking and unlinking of regions, high-performance memory copying of data between regions, and safe reassignment of an object's primary region. A list of the data manager's actions is given in Table 4.4.

[3]While this work is limited to working in the virtual address space, the data manager could be implemented at the OS or hardware level using physicial addresses as well.

Signature	Description
Objects	
`primary(object) -> Region`	Get the current primary region for `object`.
`set!(object, region)`	Set region `region` as the primary region for `object`.
Regions	
`allocate(device, bytes) -> Region`	Allocate a region of length `bytes` in the specified device.
`free(region)`	Free `region` to its source memory device.
`link!(a, b)`	Link regions `a` and `b`.
`unlink!(a, b)`	Unlink regions `a` and `b`.
`copyto!(dst, src)`	Copy memory from region `src` to region `dst`.
`sizeof(region) -> UInt64`	Return the size of region `seg` in bytes.
`getlinked(region, device) -> Union{Region, Nothing}`	Get the linked region for region `region` on `device` if it has one.
`in(region, device) -> Bool`	Return `true` if region `region` is on `device`.
`markdirty!(region)`	Mark `region` as dirty with respect to any linked regions.
`markclean!(region)`	Mark `region` as clean with respect to any linked regions.
`isdirty(region) -> Bool`	Return whether or not `region` has been marked as dirty.
`parent(region) -> Object`	Return the object to which is assigned to `region`.
Devices	
`evictfrom!(cb, device, region, size)`	Starting at `region` on `device`, free up enough space so a contiguous allocation of `size` can be made. On each encountered region on device that is not free, pass that region to a callback `cb`, after which the region will be assumed to be free.

TABLE 4.4. Base level API for a data management engine.

There are three broad categories of functions: those working on objects, those working on regions, and those working on devices. The former consists of just four functions, `primary`, to obtain the primary region for an object and `set!`, and to update an object's primary.

Functions in the second category include `allocate` and `free` methods for each supported device, as well as a fast memory copy between and within devices. Regions can be linked or unlinked using `link!` and `unlink!` respectively. The function `copyto!` provides a way copying data from one region to another. The next three functions (`sizeof`, `sibling`, and `in`) are queries for obtaining the size of a region, asking if a region has a sibling on the specified device, and querying the device to which a region belongs. Regions can be marked and queried as dirty or clean, which helps

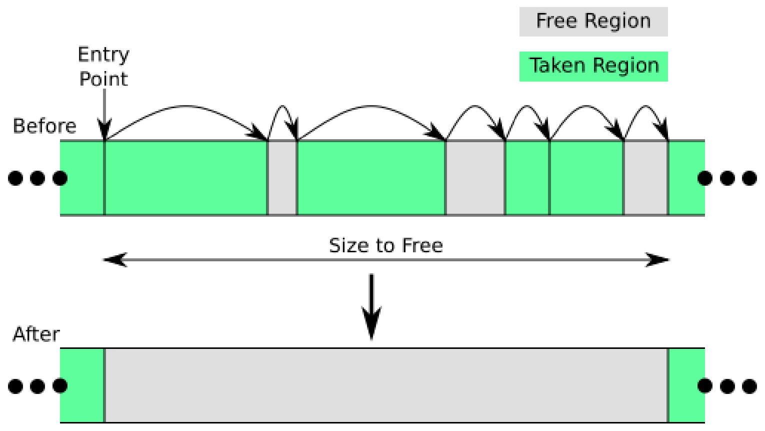


FIGURE 4.3. Illustration of the logic behind `evictfrom!`. Starting at the entry region, the function traverses each region. Free regions are gathered for free. If a region is *not* free, handling of the region is delegated to a callback. In practice, this callback implements a function like `evict!` outlined in Listing 4.1. After the callback, region processing continues. Once the requested amount of contiguous space has been collected, the spanned range is coalesced into a single contiguous free region.

maintain consistency when an object is associated with multiple regions. Finally, `parent` provides a mechanism of going from a region to its associated object.

The devices in *CachedArrays* also expose the `evictfrom!` function. As illustrated in Figure 4.3, the goal of this function is to free up enough space on the device in order to satisfy a contiguous allocation of the requested size. Beginning at the requested region, the device steps forward (or backward) across its address space. Each occupied region experienced during the walk is passed to a callback. The purpose of this callback is basically to evict the data within the region to slow memory so that after the execution of the callback, the address space for the region is available. Once enough regions have been collected, the device coalesces all the freed space into a single free region, which can be used to fulfill a future allocation. The primary reason for implementing `evictfrom!` in this callback style is that it hides the tedious details of walking through the heap from the user of the API, allowing the user to focus on correctly implementing the callback using the other API functions exposed by the data manager.

4.3.2. Policy. The last portion of our framework is the policy. Because, data management API previously described is not trivial to use, application programmers should be shielded from these low-level details through a simpler policy API. Essentially, the policy is a program written by the

Operation	Description
`alloc_fast(bytes) -> Object`	Allocate a new object in *fast* memory of size `bytes`.
`alloc_slow(bytes) -> Object`	Allocate a new object in *slow* memory.
`in_fast(object) -> Bool`	Return whether or not the primary region for `object` is in *fast* memory.
`read_use(object)`	Indicate that `object` has been used in a *read only* context.
`write_use(object)`	Indicate that `object` has been used in a *write* context.
`unsafe_free(object)`	Preemptively free `object` (before GC mark-and-sweep).
`prefetch!(object, [force = false])`	Move `object` into *fast* memory (if it isn't already). If `force == true`, than forcibly evict objects from *fast* memory if required.
`evict!(object)`	Force evict `object` from *fast* memory.
`softevict!(object)`	Prioritize `object` for LRU-based eviction.

TABLE 4.5. The policy for *CachedArrays* exposes the API above to allow applications to provide hints and annotations regarding memory location and movement. These hints can either be placed manually by the programmer or inserted as the result of a high-level compiler pass (like was the case in AutoTM). Functions dealing with the DRAM heap (`alloc_fast` and `prefetch!`) have the option to force the operation to happen. This means that the policy will evict regions from the DRAM heap in order to fulfill the request.

application or runtime programmer that uses the data management API to efficiently orchestrate data movement. In this section, we describe how one may use the data manager API to implement a simple two-level heterogeneous memory caching system.

For this example, suppose one level of this system is *fast* memory, and the other the *slow* memory. Table 4.5 shows the API exposed by the policy for managing objects at the application level. We are able to **allocate** new arrays from either memory and **free** existing arrays. The functions `read_use` and `write_use` communicate to the policy that the corresponding objects have been used. This has the potential to influence future eviction decisions by the policy based on least-recently-used (LRU) heuristics. If desired, `write_use` can be used to mark regions as dirty. We expand on the semantics of the remaining functions below:

- **prefetch!**: Cache x from *slow* to *fast* memory. This is useful if we know x is going to be heavily used in the near future. Additionally, `prefetch!` has the option to force the movement into *fast* memory to occur. When *fast* memory is full, this causes the policy to evict objects from the *fast* memory in order to successfully allocate space for x_{fast}. The choice of *which* objects to evict is left as an implementation detail of the policy. Upon caching in *fast* memory, there remains the question of what to do with x_{slow}. If we assume that the *slow* memory is not capacity limited, then it doesn't hurt to keep x_{slow} and in fact may enable some clean eviction optimizations outlined below. It is up to the policy whether to keep the backing region or deallocate it. Our policy implementation chooses to keep x_{slow}.

- **evict!**: Forcibly from *fast* to *slow* memory. If x will not be used for some time, then it may be beneficial to move it out of the *fast* memory to make space. If x already has a sibling in the *slow* memory, only a simple copy and free of x_{fast} is needed. As an optimization, the copy of x_{fast} to x_{slow} can be elided if x_{fast} is not dirty with respect to x_{slow}. Otherwise, x_{slow} needs to be allocated.
- **softevict!**: As mentioned previously, forcing operations like `prefetch` or `alloc_fast` can result in the eviction of other objects from *fast* memory. To allow the application developer to communicate program semantics to the policy, the function `softevict!` prioritizes an object for such an eviction. Unlike `evict!` which eagerly moves the objects to *slow* memory, `softevict!` requires no such immediate movement. The application programmer can use this function if they know an object will not be used for a while but might prefer that object remains in *fast* memory if possible.

We discuss two of these functions in detail using the data management API. Throughout the discussion, keep in mind that the policy maintains the following invariant: if an object has a region in *fast* memory, then this region will be the primary region for that object.

Listing 4.1 shows an example implementation of the `evict!` function. The goal of this function is to move an object from fast memory to slow memory. If the primary region is already in slow memory, then there's nothing to be done (recall the primary region invariant). Now, the object's primary region x may already have a sibling in slow memory. In this case, we will want to use this existing region. This logic can be seen on lines 4 to 9 where we first check for an existing region before allocating a new one. Line 10 shows a potential optimization: if we can track whether or not x has been modified while in fast memory (i.e., it was moved from slow to fast for a read-only operation), then we may be able to elide the expensive copy operation. Line 13 updates the object's primary region y. If a linked region existed in slow memory, we need to unlink the fast and slow

```julia
1  function PolicyAPI.evict!(policy, object)
2      x = DataAPI.primary(object)
3      if DataAPI.in(x, FAST)
4          y = DataAPI.getlinked(x, SLOW)
5          allocated = false
6          if y === nothing
7              y = DataAPI.allocate(SLOW, DataAPI.sizeof(x))
8              allocated = true
9          end
10         if DataAPI.isdirty(x) || allocated
11             DataAPI.copyto!(y, x)
12         end
13         DataAPI.set!(object, y)
14         if !allocated
15             DataAPI.unlink!(x, y)
16         end
17         DataAPI.free(x)
18     end
19 end
```

Listing 4.1. An example of building an eviction function from the *CachedArrays* data manager API. Functions belonging to the data management API are prefixed by `DataAPI`.

```julia
1  function PolicyAPI.prefetch!(policy, object, force::Bool = false)
2      x = DataAPI.primary(object)
3      if DataAPI.in(x, SLOW)
4          y = DataAPI.allocate(FAST, DataAPI.sizeof(x))
5          if y === nothing && force
6              start_region = select_via_heuristic(policy)
7              DataAPI.evictfrom!(FAST, start_region, DataAPI.sizeof(object) do region
8                  PolicyAPI.evict!(policy, DataAPI.parent(region))
9              end
10             y = DataAPI.allocate(FAST, DataAPI.sizeof(x))
11             @assert y !== nothing
12         else
13             return
14         end
15         DataAPI.copyto!(y, x)
16         DataAPI.link!(x, y)
17         DataAPI.set!(object, y)
18     end
19     return
20 end
```

Listing 4.2. Building a prefetching/caching function out of the data movement API. This implementation maintains a region in the `SLOW` memory, which can potentially accelerate eviction (Listing 4.1). Functions belonging to the data management API are prefixed by `DataAPI`.

regions (line 14). This does *not* need to be performed if the slow region was just allocated because the slow and fast regions were then never linked to begin with. Finally, we need to free x. If y already existed as a linked region, then we must first `unlink` x and y to stay consistent.

As a more complicated example, Listing 4.2 shows a possible implementation of `prefetch!`. In line 2, the primary region for the object is retrieved. Line 3 checks the device that region resides in. If the region is already in fast memory, there is nothing to be done. Line 4 tries to allocate a similar sizes region in the fast memory. If the operation fails (because fast memory is full) and the operation is forced, then the policy will forcibly free memory from the fast memory using some heuristic like LRU. This forced eviction uses the `evictfrom!` operation described previously. Following the eviction, the fast memory allocation is performed again. Data is copied (line 15), the two region are linked as siblings (line 16) and the fast memory region is assigned as the primary region (line 17).

Overall, the policy API is much simpler to understand and use than the data manager API, enabling the programmer to communicate application level semantics to the data manager to drive its decisions.

4.3.3. Summary of *CachedArrays* Benefits. *CachedArrays* enables the following optimizations which are not possible with fully transparent data management mechanisms such as NUMA and hardware caching. Importantly, these are the same optimizations which are exploited by algorithm-specific data management strategies like Sage [**26**] and AutoTM [**44**]. At a high level, *CachedArrays* tracks semantic data usage information at the object level enabling simple and straightforward policy implementations with programmer hints or fully transparent runtime control of data allocation, placement, and movement (further discussed in Section 4.4). Other benefits are listed below:

- Initially allocate data only in one specific device (e.g., fast memory). Hardware caching requires an initial movement from backing memory to the cache. Doing this greatly reduces data movement.
- Elide useless writebacks from one device to another when the data is deallocated. Section 4.7 shows this optimization significantly reduces PM writes compared to the hardware managed cache.
- Move data at a large granularity instead of at the block-level which more efficiently uses memory devices [**43**,**48**]. Section 4.7 shows that in CNN applications with a simple policy,

CachedArrays has higher average memory bus utilization than the hardware managed cache.

4.4. *CachedArrays* Implementation

We implemented a software prototype of our data management API in Julia [**7**], a high-level compiled language targeting technical computing. The main idea behind our implementation is to implement an array datatype called a `CachedArray` in Julia backed by objects. Policy hints take the form of function calls that are forwarded from a `CachedArray` to the management policy, which uses the data management API to control the segments backing the objects. Using Julia's type system and metaprogramming capabilities, we create mechanisms that allow a developer to supply policy hints to the policy in deep learning models without needing to modify the original source. Eventually, as Julia's compiler infrastructure matures, these kinds of annotations can be moved into custom compiler passes.

Section 4.4.1 describes the basic implementation of the data manager and objects allocated by the manager. Section 4.4.2 describes the array datatype that is created around allocated objects.

4.4.1. Base Implementation. The conceptual structure of our implementation is shown in Figure 4.4. ❶ At the base, we have heaps (i.e., devices) that supply memory. The figure shows two heaps representing a fast memory (e.g., DRAM) and a slow memory (e.g., PM). These heaps are responsible for allocating the segments ❷. Two regions are shown in the figure - one allocated from the fast memory and one allocated from the slow memory. These regions shown are linked as siblings ❸, which is achieved through pointers in each segment's header.

The programmer interacts with the *object* ❹. Each object contains an explicit pointer to the primary segment as well as a reference to its data controller. In this example, the primary segment is the one allocated from fast memory. To update the primary segment, the manager mutates the corresponding field of the *object*. Julia uses a generational, non-compacting, mark-and-sweep garbage collector (GC). To allow *objects* to be reclaimed by the GC, the manager maintains GC invisible (i.e., raw pointer) references to each object and its corresponding primary region. Objects, on the other hand, have GC **visible** references to their manager to ensure that the manager can be transparently reached from any object. Finalizers are used to keep the system consistent.

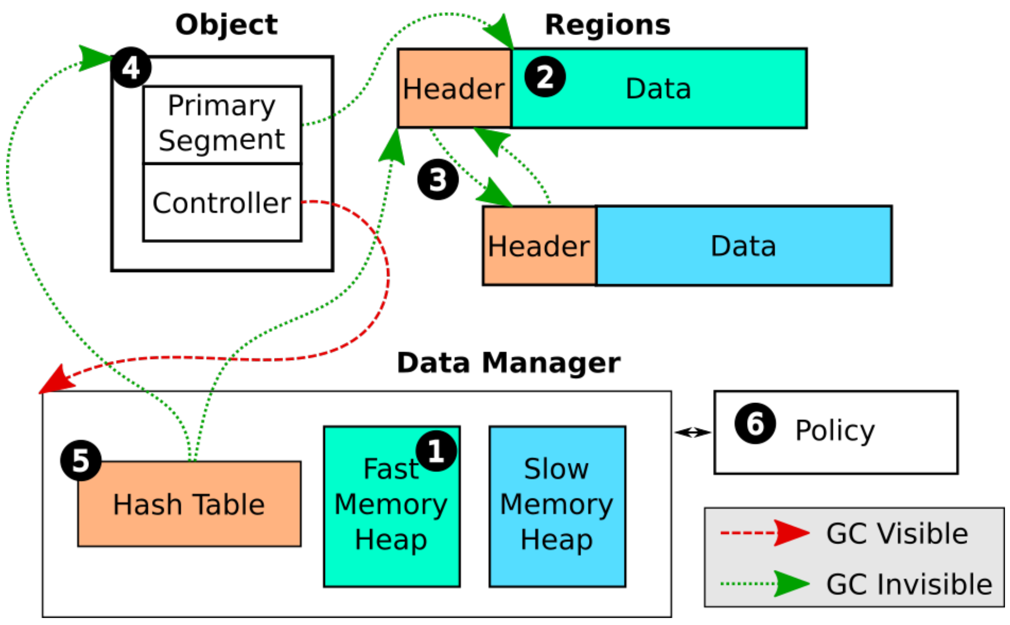


FIGURE 4.4. Overview of our the *CachedArrays*data manager in Julia. Objects are visible to the GC and implicitly backed by segments. Segments are allocated from either *fast* memory (DRAM) or *slow* memory (PM). The data manager is capable of updating the primary segment of any object. When freed by the GC, an object's primary segment is queued onto the "free buffer".

Finally, the policy ❻ is attached to the data controller to control the global behavior of all allocated objects. The policy contains LRU data structures that can be used to determine which regions should be evicted (if any) when a new region is allocated. Since objects contain references to the manager, the programmer is able to supply policy hints to objects and have those hints forwarded to the policy. To provide simple thread safety, the manager and policy are protected by a single lock.

Note that even through the policy is contained *within* the manager as an implementation detail, it still implements the policy API described in Table 4.5. Thus, the policy API is exported by the manager which then forwards requests to the policy.

4.4.2. CachedArray Implementation. In Julia, the `AbstractArray` interface is a minimal API that allows subtypes to inherit a large amount of behavior. This includes operators like matrix multiplication (including dispatch to the underlying BLAS library if the particular array type supports it). We implement a `CachedArray` data type using this interface. A `CachedArray` is a multidimensional array that can hold arbitrary plain-bits types, where a plain-bits type is either a primitive type like `UInt64` or `Float32`, or a composite struct of plain-bits types (no references to Julia allocated objects are allowed). The backing memory for a `CachedArray` is provided by an `Object` and therefore can live in either *fast* or *slow* memory. Our ultimate goal is to use `CachedArray`s to build use the policy API described in Table 4.5 and design the `CachedArray` type such that methods located deep inside a software library can be extended.

A sketch of `CachedArray`'s implementation is illustrated in detail in Listing 4.3. The definition of `Object` is given on lines 1 to 7. An `Object` contains a pointer to its primary segment and a reference back to its manager. As part of our policy integration, an extra type parameter `S` is added to the `CachedArray` type to provide finer grained read and write behavior. This parameter is one of the types `ReadWrite` (array can be both read and written), `Readable` (read only), or `NotBusy` (can neither be read nor written). These access type parameters are defined on lines 10 through 13. Conversion between these states (demonstrated on lines 44-48) is cheap (merely constructing a new wrapper `CachedArray` around the same `Object`) yet provides a mechanism by which usage of the array can be communicated to the policy. For example, transitioning an array to `ReadWrite` has the side effect of calling `write_use` on the underlying object which communicates to the policy that the object has been used (for LRU information) and marks the object's primary region as "dirty". The manager can monitor frequently accessed arrays (as determined by frequent conversions to either readable or writable) and prioritize keeping these arrays in fast memory.

Lines 16 to 19 define the `CachedArray` type, which contains an `Object` and array dimensions. Convenience type aliases are defined on lines 22 to 27 to help control function dispatch. Implementation of Julia's `AbstractArray` API occupies lines 30 to 41. Note that the read accessor method `getindex` is only defined for `ReadableCachedArrays`. This ensures that a `CachedArray` has been transitioned into the correct state (with that state communicated to the cache manager), otherwise an error will be thrown. A similar approach is used for `setindex!`.

As a usage example, Listing 4.4 shows the performance implication of matrix multiplication using *CachedArrays* with prefetch and eviction annotations. The function `mul!` perform matrix multiplication `A * B` and stores the results into `C`. Line 3 constructs the data manager, whose arguments are omitted for simplicity. Lines 4-6 construct three large matrices. Input arrays `A` and `B` are allocated directly through the `DataManager`. The destination array `C` is created based on `A`

```julia
# Object Definition
mutable struct Object{C<:AbstractController}
    ptr::Ptr{Nothing}
    manager::C
end
Base.pointer(o::Object) = o.ptr
write_use(o::Object) = write_use(o.manager, o)

# Statuses
abstract type Status end
struct NotBusy <: Status end
struct ReadOnly <: Status end
struct ReadWrite <: Status end

# Cached Array Definition
struct CachedArray{T,N,S<:Status,C<:AbstractController} <: DenseArray{T,N}
    object::Object{C}
    dims::NTuple{N,Int}
end

# Convenience aliases
const Readable = Union{ReadOnly,ReadWrite}
const Writable = ReadWrite

const ReadableCachedArray = CachedArray{T,N,<:Readable,C} where {T,N,C}
const WritableCachedArray = CachedArray{T,N,<:Writable,C} where {T,N,C}
const UnreadableCachedArray = CachedArray{T,N,NotBusy,C} where {T,N,C}

# AbstractArray API
Base.pointer(A::CachedArray{T}) where {T} = Ptr{T}(pointer(A.object))
Base.size(A::CachedArray) = A.dims
Base.IndexStyle(::Type{<:CachedArray}) = Base.IndexLinear()
function Base.getindex(A::ReadableCachedArray, i::Int)
    @boundscheck checkbounds(A, i)
    return unsafe_load(pointer(A), i)
end

function Base.setindex!(A::WritableCachedArray, x, i::Int)
    @boundscheck checkbounds(A, i)
    return unsafe_store!(pointer(A), x, i)
end

# Example Status Conversions
writable(A::WritableCachedArray) = A
function writable(A::CachedArray{T,N,S,C}) where {T,N,S,C}
    write_use(A.object)
    CachedArray{T,N,ReadWrite,C}(A.object, A.size)
end
```

Listing 4.3. Sketch of implementing a CachedArray in Julia satisfying the basic AbstractArray interface. Type parameters: `T` is array element type, `C` is the type of the manager, and `N` is the dimensionality of the array.

using the function `similar`. The arrays are moved into DRAM using `prefetch!` and we perform the matrix multiplication. Next, all arrays are moved into PM using the `evict!` function and we rerun the matrix multiplication. The execution time for the next matrix multiplication is much lower reflecting lower performance of PM compared to DRAM.

```julia
# Construct a data manager and initialize large
# matrices `A`, `B`, and `C`.
manager = DataManager(...)
A = CachedArray(randn(Float32, 10000, 10000), manager)
B = CachedArray(randn(Float32, 10000, 10000), manager)
C = similar(A)
# Move all arrays into DRAM
prefetch!(A, B, C)
@time mul!(C, A, B)
# stdout> 0.768280 seconds (1 allocation: 32 bytes)

# Move all arrays into PM
evict!(A, B, C)
@time mul!(C, A, B)
# stdout> 3.235098 seconds (1 allocation: 32 bytes)
```

Listing 4.4. Usage of `CachedArray` usage in code.

4.5. Annotations for CNN Workloads

For our DNN implementations in Julia, we implemented the back-propagation algorithm [**58**] using ChainRules.jl[4] style reverse rules (`rrule`). Briefly, an `rrule` (see Listing 4.5) is a function that returns the *primal* value `y` (i.e., result of a normal evaluation) of a function `f` and a *pullback* function. The pullback function maps the *adjoint*[5] of the output `ȳ` and returns a tuple of the adjoints of each of the function's input variables. In the case of DNN training, this pullback is a closure capturing intermediate activations. Automatic differentiation frameworks can use these pullback definitions to construct the full forward and backward passes for neural networks.

```julia
function ChainRulesCore.rrule(f::F, x...) where {F}
    ...
    return y, pullback
end
```

Listing 4.5. General signature for an `rrule`. The primal return value `y` must be equal to `f(x...)` and the pullback is a function mapping the adjoint `ȳ` to the adjoints of the arguments (`f̄`, `x̄...`).

Implementing the policy annotations is critical to achieving high performance for large scale workloads, as demonstrated in previous chapters. Our strategy for data movement and placement is taken from heuristics described by vDNN [**89**] and AutoTM [**44**]. Upon a kernel execution, we

[4]`https://github.com/JuliaDiff/ChainRules.jl`
[5]Adjoint: Partial derivative with respect to the whole network's loss variable.

```julia
function ChainRulesCore.rrule(conv::Conv, x::UnreadableCachedArray)
    # Prefetch the input tensor `x` and weight/bias tensors stored in `conv`.
    prefetch!(conv, x; force = true)

    # Execute the convolution by recursing with a readable CachedArray.
    # Marking `x` as readable will dispatch to the correct `rrule` implementation.
    # * `y`: The primal value of the convolution operation.
    # * `pullback`: Closure implementing the backwards operation.
    y, pullback = @noescape manager(x) ChainRuleCore.rrule(conv, readable(x))

    # Wrap `pullback` in another closure to annotate the backwards operation.
    function pullback_wrapper(dy)
        # Prefetch intermediate tensors `x` and `y` (lexically scoped)
        prefetch!(x, y, dy; force = true)

        # Execute the backwards kernel.
        # Variable `dargs` contains the propagated partial derivatives.
        dargs = @noescape manager(x) pullback(readable(dy))

        # Cleanup intermediate state that is no longer needed.
        cleanup!((y, dy, pullback), preserve((conv, x)))
        softevict!(conv)
        return dargs
    end

    # Prioritize for eviction if needed.
    softevict!(conv, x)
    return release(y), pullback_wrapper
end
```

Listing 4.6. Full memory annotation used for a convolution layer during VGG training.

prefetch all inputs to be used by that kernel into DRAM, ensuring that all outputs are also located in DRAM. On the forward pass, we then mark all kernel inputs as eviction candidates, meaning that if the policy experiences memory pressure these objects will be prioritized for eviction on an LRU basis. During the backward pass, we aggressively free the variables captured by the pullback closure that were used to compute the backward pass to progressively reduce memory consumption during the whole backward pass.

Listing 4.6 demonstrates a manual `CachedArray` annotation used in VGG style networks. In detail, the annotated function first prefetches its argument array `x` and the weight/bias arrays held within `conv`. The original implementation for `ChainRulesCore.rrule(::Conv, ::DenseArray)` is then manually called by converting `x` to `Readable`, which also allows the policy to infer array usage[6]. The backwards pass closure `pullback` is then annotated by wrapping it in yet another closure `pullback_wrapper`.

[6]More powerful techniques for calling the intended method are available if this approach is insufficiently expressive.

Note that the pullback wrapper also eagerly frees intermediate state that was captured by the original pullback closure (line 9) using the `preserve` and `cleanup!` utility functions. In essence, this is the freeing of intermediate state that techniques like AutoTM [**44**] and vDNN [**89**] use. Finally, both the primal and pullback implementations use `softevict!` to prioritize the argument `x` and weight/bias arrays for eviction should memory pressure arise.

Most of our annotations follow the pattern given in Listing 4.6 and are tailored to each network. For example, since VGG style networks are linear, we can call `cleanup!` after each layer on the backward pass. This does not work for networks like ResNet or DenseNet, however, due to the more complicated data flow. Instead, we create larger abstractions such as the `ResidualBlock` for ResNet and perform object cleanup on this coarser level.

4.5.1. The Need for Memory Optimizations. The example policy implementation in Listing 4.6 contains two memory lifetime related optimizations: `cleanup!` and `@noescape`. These are used to reduce reliance on the garbage collector. In particular, Julia's GC heuristics aren't well tuned for types like `CachedArrays`. As a result, allocated arrays may be kept alive much longer than needed. This is problematic when fast memory is at a premium because these arrays will likely be evicted in the future, resulting in an unnecessary and slow write-back to PM. The utility macro `@noescape` implements escape analysis to eagerly free short lived `CachedArrays` that were allocated during a function call. The function `cleanup!` frees intermediate activations that were used to compute the backwards operation. Together, these optimizations can increase memory reuse.

4.5.2. Policy Implementation Details. Here, we describe the machinery required to implement functions like `cleanup!`. The goal of these functions is to call a function (in the case of `cleanup!`, this function is `unsafe_free` from the policy API) on all `Objects` contained within their argument variables. What makes these tricky is that these functions need to work on arbitrary types. For example, in Listing 4.6 the closure `pullback` is passed to the `cleanup!` function. In Julia, closures are anonymous structs with captured variables as members of the struct. Furthermore, these closures themselves might contain more closures resulting in arbitrarily complicated type hierarchies.

To that end, we introduce the `onobjects` function shown in Listing 4.7. The main goal of this function is to recursively traverse tree-like types (no circular references) and call the provided

```
 1  # Base case - end of recursion
 2  onobjects(f::F, x::Object) where {F} = f(x)
 3  onobjects(f::F, x::CachedArray) where {F} = onobjects(f, x.object)
 4
 5  # Handle containers
 6  function onobjects(f::F, x::Union{NamedTuple,Tuple,AbstractArray}) where {F}
 7      for i in eachindex(x)
 8          onobjects(f, x[i])
 9      end
10  end
11
12  # generic fallback using reflection
13  # specialized on each type `T`.
14  function onobjects(f::F, x::T) where {F,T}
15      isbitstype(T) || iszero(fieldcount(T)) && return nothing
16      for name in fieldnames(T)
17          onobjects(f, getfield(x, name))
18      end
19      return nothing
20  end
```

Listing 4.7. Implementation sketch for `onobjects` to traverse tree-like types (no circular references) and all the provided function `f` on each `CachedArrays.Object` discovered during the traversal.

function `f` on each `CachedArrays.Object` discovered in the traversal. Lines 2 and 3 implement the base cases where an `Object` or `CachedArray` is encountered. In these cases, the underlying `Object` is passed to the function `f`. For collections like `Tuples` or `Arrays` (lines 6 to 10), each item in the collection visited. The generic implementation for composite types is shown on lines 11 to 17. This uses Julia's reflection capabilities to traverse arbitrary types. Because `Object` are mutable types (i.e., have a stable address and are tracked by the GC), they cannot be held within `isbitstypes`. This allows the optimization on line 12 which can avoid excess compilation overhead. With this utility, the function `prefetch!`, for example, can be called on all nested objects with `x` using the following: `onobjects(prefetch!, x)`. Note, however, that the example implementation in Listing 4.7 does not generate the most efficient code and more powerful techniques like Julia's `@generated` functions are required for highest performance.

More complicated utility functions are demonstrated in Listing 4.8. The function `preserve` adds all encountered `Objects` to an associative data structure. Function `cleanup!` will call `unsafe_free` on encountered objects, unless these objects are explicitly preserved. Finally, `noescape` takes a function `f` and with the help of the data manager, free all `Objects` allocated during the execution of `f` that don't escape the function call. To do this, it samples the current highest object id from

```
1   # `onobjects` implementation.
2   # Helper functions
3   function preserve(x)
4       ids = Set{Object}()
5       onobjects(o -> push!(ids, o), x)
6       return ids
7   end
8   cleanup!(x, keep::Set{Object}) = onobjects(o->(in(o, keep) || unsafe_free(o)), x)
9
10  # noescape
11  function noescape(manager, f, args...; kw...)
12      start = readid(manager)
13      result = f(args...; kw...)
14      saved = preserve((result, f, args...))
15      stop = readid(manager)
16
17      # Manually free any objects.
18      for id in start:stop
19          # Get the object from the manager using a unique `id`.
20          # If that `id` doesn't exist, then `getobject` will return `nothing`.
21          object = getobject(manager, id)
22          if object !== nothing && !in(object, saved)
23              unsafe_free(manager, object)
24          end
25      end
26  end
```

Listing 4.8. Utility functions building towards user-defined escape analysis utilities.

the manager, then calls the function `f`. All objects discovered within the result of `f`, `f` itself, and all arguments are then collected and the highest object id from the manager is sampled again. Because the object id monotonically increments for each allocation made through the manager, all intermediately allocated objects lie within the range of ids from `start` to `stop`. In lines 18 to 25, the manager is queried to determine if an `Object` with the given id exists. If so and that object isn't preserved as either an argument or return value, than the object is eagerly freed. **Note**: This assumes that (1) `noescape` is running in a single-threaded context, (2) the function `f` does not mutate its arguments, and (3) no arguments contain circular references. With engineering effort, all of these restrictions can be loosened.

4.6. Evaluation Methodology

In this section, we describe the general methodology used evaluate the performance of *CachedArrays*. Unless otherwise specified, our case studies were performed as the sole user of a 2-socket 56 core (112 thread) Intel Xeon Platinum 8276L running Ubuntu 21.10 with 192 GiB (6x32 GiB)

Model	Batchsize	Footprint
DenseNet 264	1536	526 GB
ResNet 200	2048	529 GB
VGG 416	256	520 GB

TABLE 4.6. Large scale CNN Benchmarks

DRAM and 1.5 TB (6x256 GiB) Optane DC PM per socket. Experiments were conducted on a single socket, with one thread per physical core.

To implement deep learning primitives, we wrote a Julia wrapper around Intel's oneDNN [**23**] library. OneDNN is a library the provides kernels such as convolutions, dense layers, and batch normalization that are tailored for high performance on Intel CPUs. Our wrapper library uses the oneDNN C-API to create and execute these kernels from Julia. Memory for kernel input and output tensors comes from the Julia side, and our library has been written such that this memory is backed by some suitable `AbstractArray`. This allows our library to be its own stand-alone library independent of *CachedArrays* while still offering full support for its memory management capabilities. More details about this wrapper library are provided in Appendix A.

4.6.1. Large Networks - Comparison With 2LM. The Xeon server used to run these experiments can be configured in two modes, allowing multiple different use cases for PM. The *memory mode* (2LM) configures PM as main memory with DRAM serving as a transparent, direct-mapped cache [**43**]. *App direct* allows PM to be used directly. In this mode, PM can either be configured as an extra NUMA node to be used automatically by the OS, or mounted as a direct access (DAX) file system. In the latter case, files memory mapped from the PM-based DAX file system are directly mapped into the program's address space with reads and writes sent directly to the PM devices. Our full *CachedArrays* based policy uses this last option.

To compare against 2LM, the overall memory footprint of the models used for benchmarking needs to greatly exceed the size of DRAM cache on a single socket. This was achieved though a combination of deep networks and large batchsize. For large traditional networks, we used ResNet 200 [**41**] and DenseNet 264 [**46**], two networks with complex data flow. As an extra comparison point, we implemented VGG 416 [**89**], which is essentially a greatly extended VGG 16. A summary is provided in Table 4.6 along with the approximate minimum memory footprint required for a single iteration of training.

Furthermore, we want to investigate the relative impact of the various optimizations described in Section 4.5. These optimizations include:

- **Memory Optimizations (M):** Section 4.5 describes memory optimizations like `cleanup!` and `@noescape` to free memory as soon as possible. By doing this, we reduce the amount of data kept alive longer than necessary by the garbage collector. If this intermediate data is kept alive, than it must be written to PM if evicted to maintain correctness, resulting in unnecessary slow PM writes. Disabling this optimization means we need to rely on the garbage collector for resource management which involves explicitly triggering collection when memory pressure is detected.
- **Local Temporary Allocations (L):** As discussed in Section 4.2.3, *CachedArrays* has been designed to support unlinked regions in fast memory. In essence, this allows newly created arrays to be allocated in DRAM only as opposed to a true cache which would always have a backing region in PM. In combination with the memory optimizations, this is a potent tool for reducing PM traffic. Our policy can be modified to disable local allocations, in which case a newly created array *must* first be allocated in PM and then copied into DRAM before use. The purpose of disabling this optimization is to more directly compare with a naive 2LM implementation. Since 2LM operates as a DRAM cache, each physical cache line must ultimately have a copy in PM. By effectively generating a compulsory miss on first access with *CachedArrays*, we can more closely model the behavior of 2LM.
- **Prefetching (P):** The discussion in Section 4.5 discussed prefetching any arrays that would be used during kernel execution. However, Figure 3.4 suggests that kernel execution time may be less sensitive to the location of read-only arguments. Further, prefetching requires making room in DRAM for the prefetched arguments, which could result in other arrays being evicted to PM. To explore this trade-off, we also prefetching to be disabled globally.

Abbreviation	Description
2LM: ∅	2LM with no memory optimizations. Garbage collector will be invoked when the heap reaches 80% utilization.
2LM: M	2LM with memory freeing optimizations. Invocation of the garbage collector is not needed.
CA: ∅	*CachedArrays* with no memory optimizations or prefetching. Furthermore, *all* arrays begin in PM and must be moved into DRAM before use, mimicking what would happen in a true cache. Like "2LM: ∅", the GC will be invoked if the PM heap reaches 80% utilization.
CA: L	*CachedArrays* with no memory optimization or data prefetching. However, unlike "CA: ∅", arrays can be allocated in DRAM only and don't need a linked region in PM.
CA: LM	*CachedArrays* with memory optimizations but no data prefetching.
CA: LMP	*CachedArrays* with memory optimizations *and* prefetching.

TABLE 4.7. Combinations of operating modes and optimizations explored for the very large CNN models.

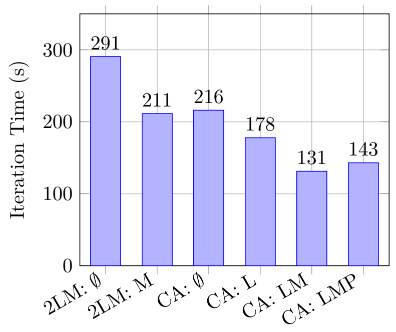
The combination of optimizations and operating modes is outlined in Table 4.7. Comparing "2LM: ∅" and "2LM: M" will show the impact that the higher virtual memory reuse induced by the memory optimizations has on the performance of the DRAM cache. The *CachedArrays* based runs will then show how *CachedArrays* compares against 2LM as well as impacts of various optimizations.

All runs in 2LM were conducted with a maximum memory size of 1300 GB. When we run in *app direct* mode, *CachedArrays* is configured with the same hardware limits of 180 GB DRAM and 1300 GB PM. After each training iteration (forward + backward pass), the GC was invoked to clean up all temporary memory, leaving only the model weights and computed gradients. The local heap was than defragmented before the next run to help keep behavior similar across iterations (defragmentation overhead is negligible compared to the iteration time). Each model was run for 8 iterations and performance metrics were checked to ensure that behavior for each iteration was consistent. Input data was randomly generated using a normal distribution.

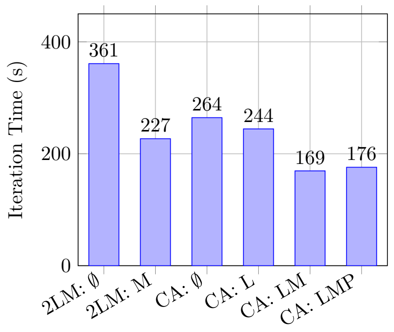
For each experiment, we used hardware performance counters to capture read and write traffic to DRAM and PM. For the 2LM based runs, the same hardware counters were used to also capture DRAM cache statistics including cache hits, clean cache misses, and dirty cache misses. All memory heaps used by *CachedArrays* are pre-allocated before running our experiments, ensuring that the OS assigns physical pages to all virtual pages within the heap. This in itself provides a large speedup over Julia's default allocator. For large allocations, normal allocators typically memory

Model	Batchsize
DenseNet 264	504
ResNet 200	640
VGG 116	320

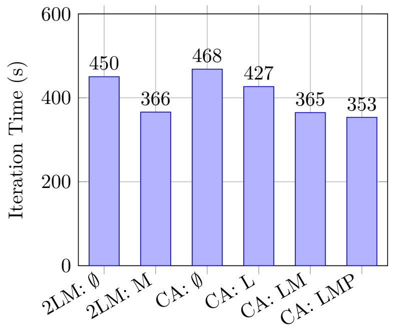
TABLE 4.8. Small scale CNN benchmarks for sensitivity analysis. The memory footprint for these networks is between 170 and 180 GB.



(a) DenseNet 264



(b) Resnet 200



(c) VGG 416

FIGURE 4.5. Average execution time for a single iteration of training for the large networks, categorized by operating mode and applied optimizations. Refer to Table 4.7 for a description of the x-tick marks.

map new more memory from the OS, which must be zero-initialized and the virtual to physical address translation established. Because of this overhead, we do not present results with Julia's default allocator and instead use 2LM with the *CachedArrays* allocator as the baseline.

4.6.2. Small Networks - Sensitivity Analysis. As with AutoTM, we can vary the amount of DRAM available to *CachedArrays* to see how our simple policy holds up as we decrease the DRAM allowance. To that end, we use the networks and batch sizes provided in Table 4.8. These are chosen such that the memory footprint required for training fits within 180 GB and thus can fit within the DRAM of a single socket of our benchmark machine. We then vary the DRAM budget from the full 180 GB down to 0 GB (PM only). Note that for this experiment, we use VGG 116 instead of VGG 416 in order to mantain a reasonably high batch size. All of these runs were conducted in the "CA: LM" mode as this tends to perform well across all networks.

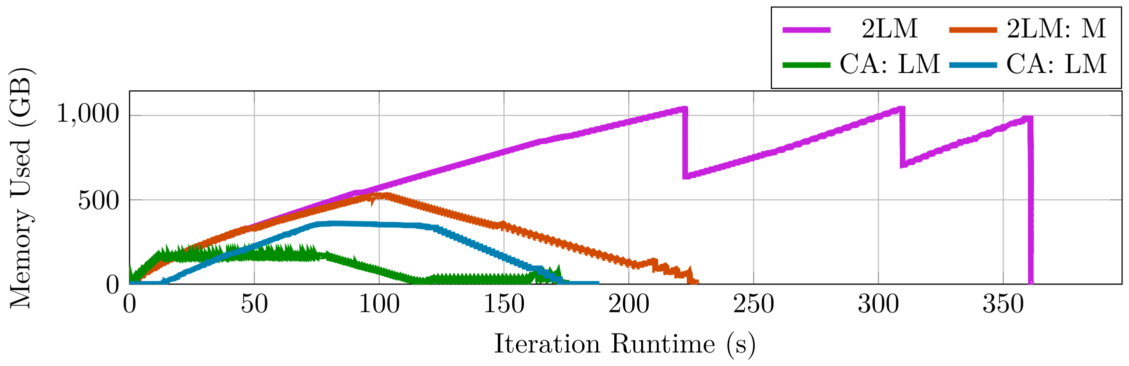


FIGURE 4.6. Resident heap memory through a single iteration of ResNet training. The 2LM based experiments have a single memory heap which is implicitly managed by Intel's DRAM cache. Full *CachedArrays* explicitly managed two heaps, one for DRAM (local) and one for PM (remote).

4.7. Results

Figure 4.5 shows the absolute runtime for a single iteration for the large CNN benchmarks. First, we explore the performance of "2LM: ∅" (2LM with no optimizations) and "2LM: M" (2LM with memory optimizations). Memory optimizations improve the performance of 2LM by 1.38×, 1.59×, and 1.23× for DenseNet 264, Resnet 200, and VGG 416 respectively. For *CachedArrays*, "CA: L" (supporting local-only allocation) is faster than "CA: ∅", and applying memory optimizations further improves performance. Prefetching hurts performance for DenseNet and ResNet, but improves performance for VGG. In summary, the fastest version of *CachedArrays* is 1.38×, 1.34×, and 1.04× faster than the fastest version of 2LM.

Two behaviors lead to *CachedArrays*' performance improvement: memory reuse distance and traffic shaping. To understand the former, consider Figures 4.6 and 4.7. Figure 4.6 shows the occupancy of the memory heaps for a single iteration of ResNet under three operating regimes. The two 2LM based runs only have a single memory heap that is implicitly managed by the hardware DRAM cache while *CachedArrays* (with memory optimizations and local allocation) has two heaps: one DRAM based and one PM based. Without any memory optimizations ("2LM: ∅"), memory usage keeps increasing until the GC is run (around time 220 s) which causes the monotonically increasing behavior of that curve. In contrast, when the annotated run ("2LM: M") begins its backward pass, (around time 100s) it proactively frees memory produced on the forward

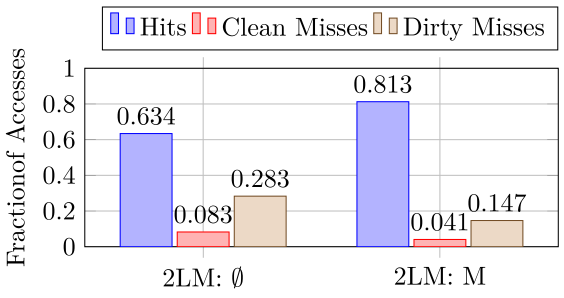


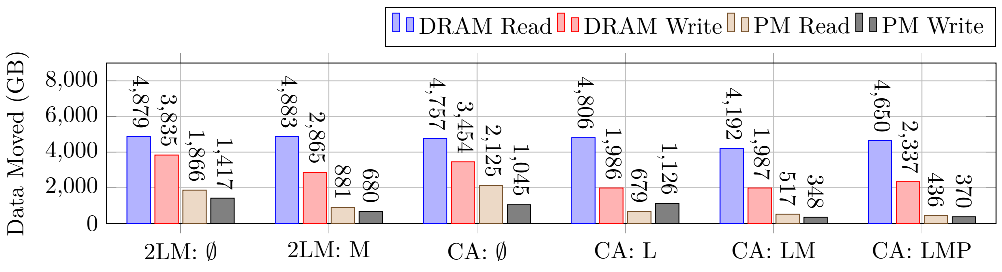
Figure 4.7. Tag statistics when running in memory mode for a single iteration of training ResNet 200.

pass. This results in more reuse of the physical pages backing that memory, leading to fewer cache misses and less data movement.

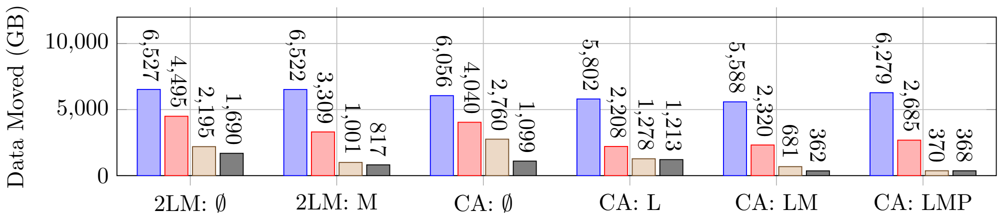
Supporting this idea is Figure 4.7, which shows the average DRAM cache hit, clean miss, and dirty miss rates for the two 2LM runs. The annotated run has a 18% higher hit rate 50% lower dirty miss rate, both of which improve 2LM performance [**43**]. By adding semantic information, we achieved better utilization of the underlying data movement mechanism.

To understand the performance difference between 2LM and *CachedArrays*, we also need to understand the total amount of memory moved for a single iteration of training. Figure 4.8 shows the total amount of DRAM and PM traffic for a single iteration of training for each of our large networks, further broken into reads and writes. With no optimizations, *CachedArrays* is slower than memory-optimized 2LM and in the case of VGG is even slower than unoptimized 2LM. For DenseNet and ResNet, "CA: ∅" generates similar read and write traffic to "2LM: ∅", though with generally fewer PM writes. The saving of PM writes occurs because even though memory optimizations are not applied, we still run the garbage collector after every iteration of training. In 2LM, this does not really help because physical addresses used on the backwards pass are still dirty with respect to the DRAM cache (see the discussion in Section 2.5.2). However, this *does* help *CachedArrays*, resulting in better performance.

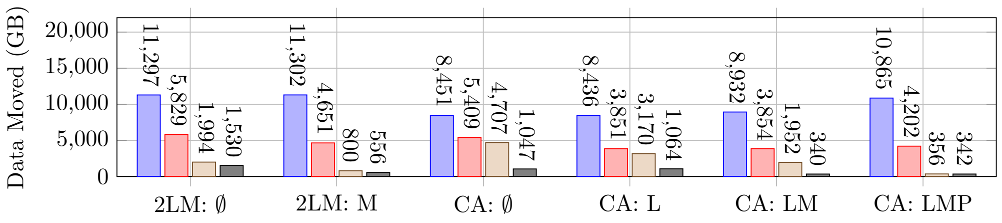
Even through this still applies to VGG, "CA: ∅" is still slower than "2LM: ∅". This is where *traffic shaping* comes into play. From Chapter 2, we know that large sequential accesses provide the highest bandwidth for PM. In 2LM, PM traffic is haphazard and results from conflict misses



(a) DenseNet 264

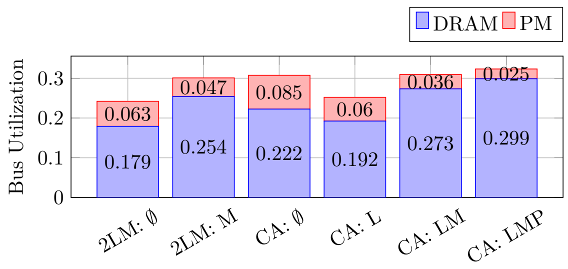


(b) Resnet 200

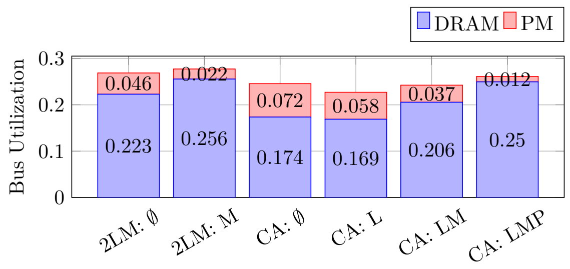


(c) VGG 416

FIGURE 4.8. Amount of data moved for a single iteration training for the large CNNs.



(a) ResNet 200.



(b) VGG 416.

FIGURE 4.9. Average utilization of the six channel 2.666 MT/s DRAM bus.

in the DRAM cache. With *CachedArrays*, PM traffic is the result of explicit, well shaped memory copies.

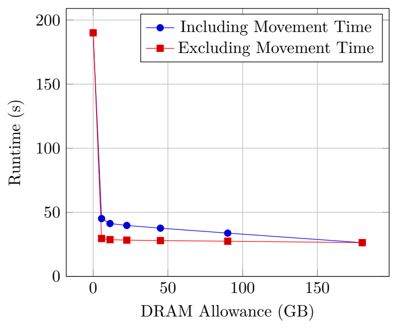
Figure 4.9 shows the average utilization of the memory bus for a single iteration of training for VGG 416 and ResNet 200. For ResNet, "CA: ∅" achieves a higher average utilization than "2LM: ∅"

while the situation is reversed for VGG. The memory movement engine in *CachedArrays* is highly multithreaded, specifically targeting large memory sizes. This works great for ResNet because the large batch size of 2048 results in large memory transfers. However, a much smaller batch size of 256 is used for VGG, leading to smaller data transfers and more parallelization overhead. Note, however, that bus utilization isn't the whole story and must be considered in the context of overall memory moved. As optimizations are applied via *CachedArrays*, bus utilization tends to increase and overall all traffic generated tends to decrease, both resulting in better performance.

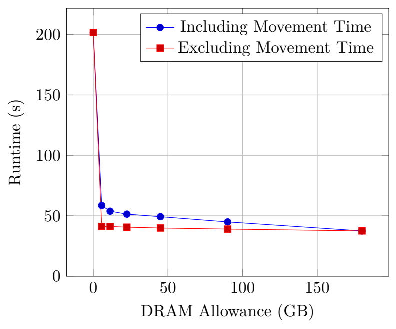
Impact of Local Allocation: Adding the local allocation optimization to *CachedArrays* significantly decreases the PM read and DRAM write traffic (due to the elision of the initial memory copy). The performance difference between these two is largely due to the decreased time spent synchronously moving data.

Impact of Memory Optimizations: Memory optimizations decrease memory pressure by freeing memory as soon as possible. This avoids many unnecessary writes to PM, which can be seen in Figure 4.8. In particular, observe the difference in PM reads and writes in Figure 4.8a between "CA: L" and "CA: LM". For "CA: L", the number of PM writes exceeds the number of PM reads, implying that *unnecessary* data is being moved to PM. This is the result of intermediate allocations not being freed as soon as possible. When applying memory optimizations ("CA: LM"), the number of PM writes for DenseNet drops from 1100 GB to 350 GB, with PM reads exceeding PM writes. The other networks (Figures 4.8b and 4.8c) experience similar decreases in PM writes when applying memory optimizations. The local allocation and memory optimizations reduce the amount of PM writes down to a bare-minimum.

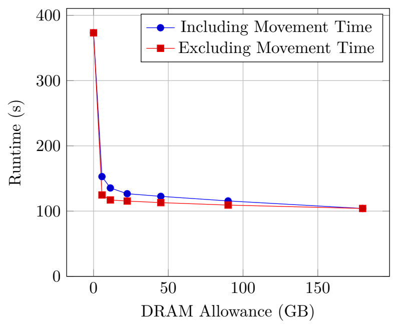
Impact of Prefetching: Enabling data prefetching harms performance for DenseNet and ResNet. As can be seen in Figure 4.8a and 4.8b, prefetching decreases PM read traffic and increases DRAM read traffic because arrays are moved from PM to DRAM where there are referenced multiple times to compute the backwards pass. However, as indicated in Figure 3.4, some operations are not particularly sensitive to the bandwidth of their read-only arguments. Hence, this prefetch wastes time and potentially evicts other arrays. In the case of VGG, on the other hand, prefetching *does* slightly improve performance since it significantly decreases PM read traffic by a factor of 5.4×. This shows that there is no "one size fits all" approach to memory management.



(a) DenseNet 264.



(b) ResNet 200.



(c) VGG 116.

FIGURE 4.10. Average runtime for a single training iteration for the small CNNs. Results are shown for both absolute wall clock time (blue) and projected time if data movement could be perfectly hidden behind computation (red). These runs were conducted with local allocation and memory optimizations, but *without* prefetching enabled.

In summary, full *CachedArrays* results in less DRAM and PM traffic overall, because *CachedArrays* is aware that data freed on the backwards pass is semantically dead, and thus does not need to be written back to PM. Hardware caches do not have this semantic insight and thus must always act conservatively. Furthermore, Figure 4.8 shows the average DRAM bus utilization though out an iteration of training. *CachedArrays* maintains a higher average utilization while also moving less total data. With *CachedArrays*, we are able to both maintain the memory semantics required to elide unnecessary dirty writebacks and use traffic shaping achieve high bandwidth.

4.7.1. Results for Small Networks. The runtime for a single iteration of training for the smaller networks is shown in Figure 4.10. As with AutoTM, running with only PM results in a 3–4× performance penalty. However, with even just a little DRAM, we're able to get most of that performance back. These results are consistent across the models used.

Figure 4.10 also shows what the performance would be if *CachedArrays* had perfectly asynchronous data movement (as opposed to purely synchronous) and could overlap movement with execution. One possible implementation of this asynchronous data movement would be to keep two distinct thread pools, one for compute and one for data movement, and launching both kinds of operations concurrently. For DenseNet and ResNet, this projected performance varies only slightly as the DRAM budget decreases. VGG, on the other hand, still experiences a slow-down due to more and more reads being generated to PM. These results are consistent with the large network results where it was observed that DenseNet and ResNet had lower performance with prefetching while VGG's performance improved with prefetching. Evidently, the kernels composing VGG are more sensitive to read bandwidth.

To understand why only a small amount of DRAM is needed for a large performance improvement over all PM, refer to Figure 3.3 which, amongst other things, shows the scaling of DRAM to PM copy bandwidth with threads. Contrary to the behavior of pure DRAM, DRAM to PM copy bandwidth actually *decreases* with increasing threads. Furthermore, the copy kernel implemented in Figure 3.3 uses *nontemporal* stores to PM, which are crucial for best performance[7]. OneDNN kernels are *not* optimized for writing to PM and as such result performance with all PM is slow[8]. When a small amount of DRAM is used, than the output parameters of the computation kernels can placed in DRAM and any movement of data from DRAM to PM goes through code-paths optimized to get the best write performance out of PM. An interesting area for future research would be to explore computation kernel implementations that specialize based on the memory location of its arguments (much like the specialization via just-in-time compilation based on the dimensions of the arguments).

The above results demonstrates that while simple policies can achieve good performance, there is still room for more advanced memory management policies like AutoTM.

4.8. Related Work

Table 4.1 presented a high level summary of the work closely related to this paper. Data tiering at the OS level uses intelligent page migration [**27**, **28**, **53**, **55**, **86**] to move pages between slower and faster memories based on heuristics such as hotness and probability of reuse based on runtime characteristics. Techniques such as Nimble Pages [**108**] and KLOC [**54**] explore different page granularities, efficient page swapping, and tailoring page migration to specific data structures such as kernel objects. However, none of these workloads address deep learning workloads, especially those with very large memory footprints and sparse embeddings which are the focus of this work.

[7]Standard stores require a read from PM for cache-coherence reasons, resulting in a mix of read and writes to PM.
[8]Recall that Optane DC (PM) is primarily bandwidth limited during writes.

For deep learning workloads there has been a plethora of work in the CPU/GPU heterogeneous environments to overcome the memory limitations of GPUs. Works like vDNN [**89**] and its derivatives [**15**, **83**, **85**] exploit the unique characteristics of the backpropagation algorithm. These works were generalized in frameworks such as AutoTM [**44**] that formalize the optimal tensor placement and movement problem in the backpropagation algorithm using mathematical optimization techniques, and Sentinel [**87**] by taking advantage of the runtime profiling information of PM/DRAM based heterogeneous memory systems. However, these techniques cannot be directly applied to workloads such as DLRM that have sparse embedding tables and workloads where the reuse patterns are less straightforward. Sage [**26**] explores this problem for large scale graph analytics. The authors propose new data structures and algorithms that partition the data so that writes are directed to DRAM while read heavy data is stored in PM. In contrast, *CachedArrays* is applicable to both "simple" backpropagation algorithms and sparse accesses such as embedding tables (covered in the next chapter) by allowing *specialization of the policy.*

4.9. Discussion

This chapter presented an API and abstraction layer that can facilitate algorithmic development using heterogeneous memory. However, policy implementation is largely left to the programmer, whether through direct injection of commands like `prefetch!` or through customization of the policy component of the cache manager. The example policy used for the CNN case study has been studied by works like vDNN [**89**] and AutoTM [**44**] and as such, this chapter focused more on demonstrating the mechanism rather than the policy. In this section, we will explore techniques that can build on top of `CachedArrays` to automatically implement policies. As an example, we will sketch how one might implement AutoTM on top of our `CachedArrays` and `OneDNN` libraries.

In Julia, the relationship between compile-time and run-time is blurrier than many static languages such as C++ or Rust. Much of the Julia compiler is itself written in Julia and can be invoked and inspected at runtime. Staged compilation features such as the `@generated` function allow function bodies to be programatically generated. This has lead to packages like `Cassette.jl` [**88**] and `IRTools.jl`[9] which allow the user to inject context-specific transformations into the Julia

[9]`https://github.com/FluxML/IRTools.jl`

```julia
using Cassette
Cassette.@context MyCtx
Cassette.prehook(::MyCtx, f, args...) = println("MyCtx: ", f, typeof.(args))

# Test Function
inc(x) = x + 1
mul(x, y) = x * y
inc_double(x) = mul(inc(x), inc(x))

# Run the function normally.
inc_double(10)
# stdout> 121

# Run under Cassette
Cassette.overdub(inc_double, 10)
# stdout> MyCtx: inc(Int64,)
# stdout> MyCtx: +(Int64, Int64)
# stdout> MyCtx: add_int(Int64, Int64)
# stdout> MyCtx: inc(Int64,)
# stdout> MyCtx: +(Int64, Int64)
# stdout> MyCtx: add_int(Int64, Int64)
# stdout> MyCtx: mul(Int64, Int64)
# stdout> MyCtx: *(Int64, Int64)
# stdout> MyCtx: mul_int(Int64, Int64)
# stdout> 121
```

LISTING 4.9. Example of creating a contextual code transformation using Cassette.jl [**88**]. In this example, our custom context `MyCtx` simply prints the name and argument types of each sub-function called using the `prehook` mechanic (a function called on a function `f` and arguments `args` before recursively calling `f(args...)`). Note that the implementation of `inc_double` does not need to know of the existence of Cassette in order to be modified by Cassette.

compiler. An example of this is shown in Listing 4.9, which shows a simple method for printing out all lower-level functions called to implement the top function `inc_double`.

A more powerful version of the toy example provided in Listing 4.9 is implemented by the package `Ghost.jl`[10]. This package creates a linearized Wengert list [**5**] for a given function. This list can then be inspected, modified, and compiled. An example of a simplified trace for Vgg19 using the `OneDNN` framework is shown in Listing 4.10. Here, we can see the function calls for individual convolutions (e.g. lines 4 and 8), pooling (line 11), and the final dense layers (lines 26 to 35).

To implement AutoTM, one could inspect this list, profile intermediate kernels with inputs in combinations of DRAM and PM (using CachedArrays), One could then construct tensor graphs by inspecting the tape, perform the ILP optimization, modify the tape to insert the appropriate prefetch and eviction instructions, and finish by compiling the tape.

[10]`https://github.com/dfdx/Ghost.jl`

```
...
%9 = getfield(%6, 1)::OneDNN.Conv{..., typeof(relu), 2}
%10 = getfield(%6, 2)::Int64
%11 = %9(%2)::Memory{Float32, 4, Array{Float32, 4}}
...
%15 = getfield(%12, 1)::OneDNN.Conv{..., typeof(relu), 2}
%16 = getfield(%12, 2)::Int64
%17 = %15(%11)::Memory{Float32, 4, Array{Float32, 4}}
...
%21 = getproperty(%3, maxpool)::Pooling{OneDNN.Lib.dnnl_pooling_max, 2}
%22 = %21(%17)::Memory{Float32, 4, Array{Float32, 4}}
...
%149 = first(%147)::OneDNN.Dense{..., typeof(relu)}
%150 = %149(%144)::Memory{Float32, 2, Matrix{Float32}}
%151 = tail(%148)::Tuple{OneDNN.Dense{..., typeof(relu)}}
%152 = first(%148)::OneDNN.Dense{..., typeof(relu)}
%153 = %152(%150)::Memory{Float32, 2, Matrix{Float32}}
%154 = tail(%151)::Tuple{}
%155 = first(%151)::OneDNN.Dense{..., typeof(relu)}
%156 = %155(%153)::Memory{Float32, 2, Matrix{Float32}}
const %157 = logsoftmax::typeof(OneDNN.logsoftmax)
%158 = %157(%156, 1)::Memory{Float32, 2, Matrix{Float32}}
```

LISTING 4.10. Truncated, linearized execution tape for Vgg19 using OneDNN generated by the Ghost.jl package. The tape is represented as a Static Single Assignment (SSA) representation.

Now, this implementation sketch ignores many engineering details required to actually make such a system work and robust. Nevertheless, it demonstrates a feasible path towards implementing automatic heterogeneous memory management using CachedArrays as a building block, rather than modifying an existing framework to be aware of heterogeneous memory.

4.9.1. Enabling Dynamic Policies. The policies explored in this chapter were fixed. That is, annotations like `prefetch!` where either universally enabled or disabled. However, as seen in Figure 4.5, some networks benefit from prefetching while others don't. A more powerful implementation might be able to make these decision on the fly using online profiling.

AutoTM used offline profiling of kernels to drive its mathematical decision making. A similar technique could be used for online profiling. For instance, every time a kernel was executed, the policy could be notified. During the profiling phase, the policy could explore different combinations of argument locations in DRAM and PM, building an online profile of kernel behaviors. This behavior could then be used to drive future decisions about whether to prefetch or not (e.g., kernels that aren't sensitive to the location of their arguments wouldn't require prefetching).

In the context of Julia, proper implementation would require care. OneDNN kernels could opt-in to this kind of profiling through modifications to the source code of the wrapper library. However, operations can be implemented in pure Julia as well. In the latter care, we can again use compiler transformations like Cassette to annotate specific kernels as “profile points”. The function dispatch tuple (i.e., for a function `f(args...)`, its dispatch tuple is `Tuple{typeof(f),typeof.(args)...}`) as well as properties of the arguments (such as size of any arrays occurring in the arguments) could be used to as a key for the profile table. However, such profiling must be done at an appropriate granularity to avoid unnecessary overheads.

4.9.2. Limitations. Our *CachedArrays* prototype was implemented in user space and while it provides significant performance improvements, there are limitations. For instance, on current hardware, memory copying must be performed using the CPU cores, removing clock cycles from compute. Our prototype also only supports ML algorithms written in Julia.

4.9.3. Generalizing to Other Languages. Languages with more aggressive memory management like C++ or Rust may have simpler implementations and may require fewer manual memory annotations. We can implement *CachedArrays* in the STL containers in C++ using the allocator interface. Python based frameworks like PyTorch or Tensorflow have robust compiler and runtime infrastructures that allow memory optimization passes and custom allocators.

CHAPTER 5

DLRM Case Study

5.1. Introduction

In the last chapter we introduced *CachedArrays*, a data-tiering API and implementation as a Julia abstract array data type. Using this datatype, we were able achieve similar heterogeneous memory management results as AutoTM using a bottom-up approach, rather than a top-down. In this chapter, we will again use CachedArrays and the data management ideas developed in the previous chapter, but targeting a Deep Learning Recommendation Model (DLRM) workload instead. Specifically, we will target the embedding table lookup and gradient descent update operations. Unlike CNNs which have a predictable access pattern to large dense arrays, these embedding table operations are characterized by sparse, low-volume, data-dependent accesses.

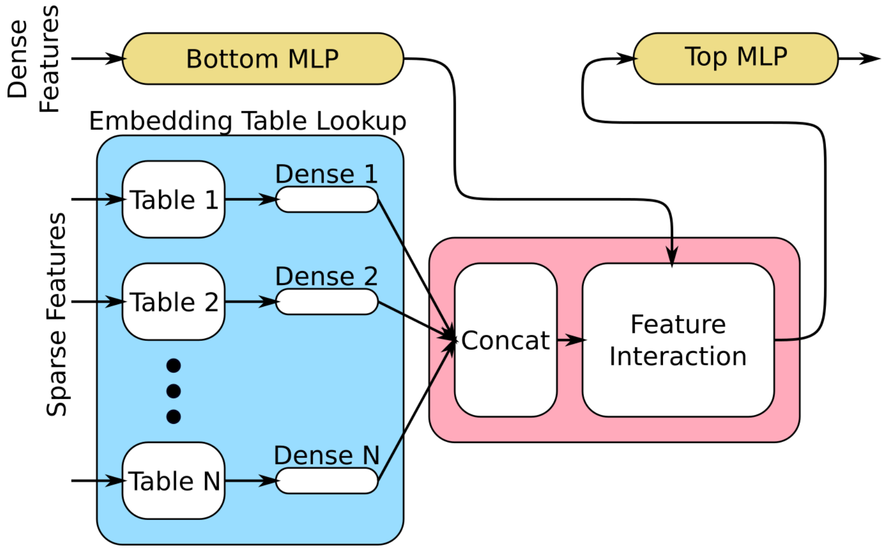


FIGURE 5.1. Generalized DLRM architecture, taking a mix of dense and sparse input features. Sparse features are converted to dense features using embedding tables. Output of the bottom MLP and embedding lookup are mixed and finally processed by a final top MLP.

The general DLRM architecture shown in Figure 5.1 is used by Facebook and other companies to serve recommendations (e.g. link recommendations, movie recommendations, or ads) to users [**67**]. As shown in the figure, the model operates on a collection of dense features and sparse features. Dense features are processed by a standard Multi-Level Perceptron (MLP) network. The sparse features, on the other hand, are used to index into embedding tables to extract dense features. Sparse features can encode information such as a user id, product id, etc. The outputs of the individual embedding table lookups are concatenated together and combined with the output of the bottom MLP using various feature interaction techniques. Post interaction tensors are processed by a final top MLP before yielding a final result.

The architectural implications of these networks has been investigated in depth in the literature [**37**]. Embedding table lookup and update operations are memory bandwidth intensive while the dense MLP layers, on the other hand, are compute intensive. This combination stresses many architecture subsystems. Further complicating matters is the size of these embedding tables, which can occupy tens to hundreds of gigabytes and are expected to grow [**37**, **52**]. In essence, these embedding table operations require fast, random access to small regions of data (ranging from 64 B to 1024 B) originating from a large pool.

DLRM style models are not the only application for embedding tables. Many other popular deep learning models like transformers [**104**] use such tables as well.

Naive methods for heterogeneous memory embedding table management may fall short for several reasons. First, just placing the tables in PM will not yield good performance if the tables are large due to the significantly lower performance of PM when compared with DRAM. Next, the reuse pattern of entries within an embedding table can vary significantly from essentially random highly local and can change over time [**31**]. This suggests the need for a dynamic policy that is capable of meeting these different requirements.

Researchers have investigated using heterogeneous memory to store portions of these embedding tables [**31**]. However, these works tend to focus on using NVMe SSDs for their tiered storage. The main issue with simple caching is that embedding table are sparsely accessed and lookups have essentially no spatial locality and varying temporal locality. In this chapter, we will discuss our high-performance implementation of DLRM in Julia, including an extendable abstract API for embedding table lookup and update, and compare our implementation with a state of the art PyTorch implementation from Intel [**52**]. We also propose applying our memory management framework to the embedding table lookup and update operations, modifying our definition of "object" to become an entry in the embedding table and taking advantage of the byte level access granularity offered by Optane DC.

The contributions of this chapter are as follows. In Section 5.2, we present a high-performance implementation of the embedding table lookup and stochastic gradient descent operations. Section 5.3 benchmarks the performance of our implementation in both DRAM and PM, demonstrates the benefits of certain optimizations, and illustrates certain performance pitfalls like overfetching by hardware performance counters. Section 5.4 introduces *CachedEmbeddings*, our approach to fine-grained heterogeneous memory management for embedding table operations and compares its performance with standard embedding tables. We then discuss our high-performance DLRM implementation in Section 5.5 and finally study the performance of *CachedEmbeddings* in the context of end-to-end DLRM training in Section 5.6.

5.2. Embedding Table Implementation

In this section, we will discuss the details of the embedding table and stochastic gradient descent (SGD) update operations, characterizing some performance optimizations and traps on real hardware.

5.2.1. Embedding Table Lookup and Update. Figure 5.2 shows a simplified example of a reducing embedding lookup and update. The embedding table M consists of a number of feature vectors (5 in the case of the example figure). The feature vectors are all of uniform length n (ranging from as low as $n = 16$ to $n > 256$) and consist of uniform primitive types (typically `Float32` or `BFloat16` [**31**]). Typically, feature vectors are n-dimensional encodings of categorical data (e.g., words in a dictionary or users of an application) derived using a process like word2vec [**65**]. In this discussion, we assume embedding tables are laid out in row-major order. That is, each feature vector in the embedding table is interpreted as a row in the embedding table.

During the lookup operation ❶, each row in the output matrix A is constructed by summing together multiple rows from the input table M in a gather operation. The indices in M to be

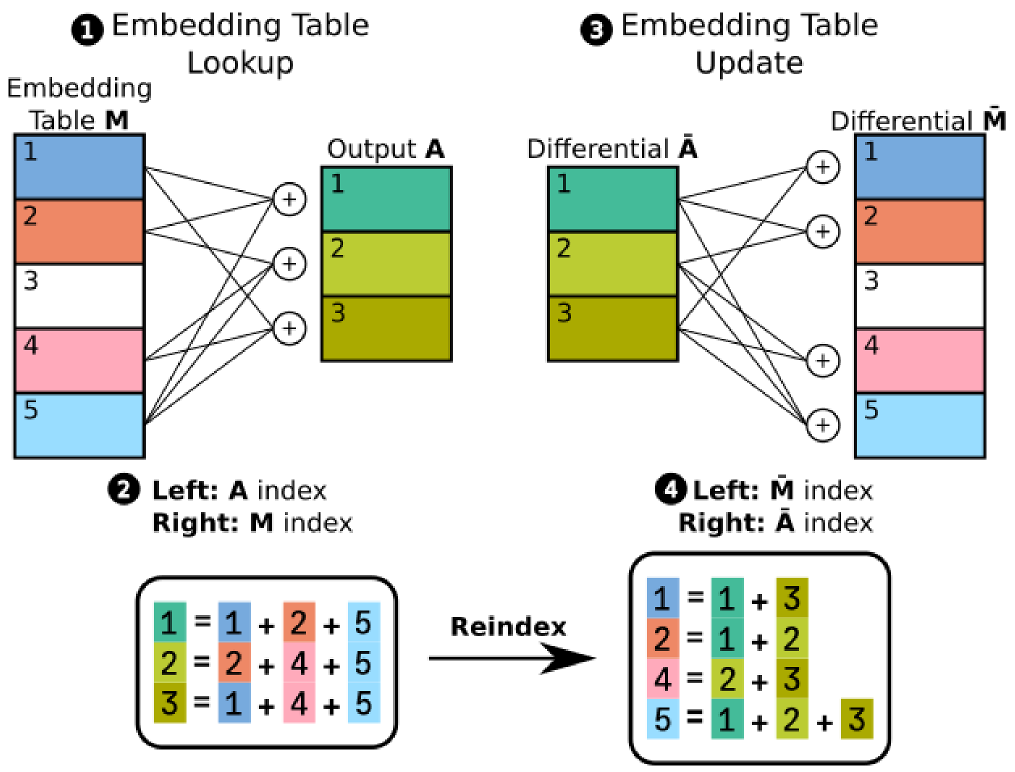


FIGURE 5.2. Example of embedding table lookup and updates. In the lookup step, entries in the embedding table are accessed, accumulated, and written to the destination. In the update step, entries adjoint $\overline{A}$ are accessed and accumulated. For an embedding table row j, the gradient consist of all indices i in $\overline{A}$ such that j was used to compute the ith entry of the primal A. All indices are in index-1 encoding.

summed ❷ can be encoded as a matrix, vector of vectors, or logically similar container. For example, to compute A_3, we perform the operation $A_3 = M_1 + M_4 + M_5$.

The pullback computation ❸[1] is like the forward computation in reverse where the table adjoint $\overline{M}$ is computed by summing together rows in the adjoint $\overline{A}$. Conceptually, a row i in $\overline{A}$ is part of the sum to compute row j in $\overline{M}$ if and only if M_j was used to compute A_i.

We use a reindexing procedure to transform the lookup indices into an collection of update indices ❹. Internally, the reindexed indices are stores in a CSR-like format and the reindexing operation itself is done in three major steps. First, a histogram is computed, mapping each seen index j of M to its count and the order in which it was seen. Second, a prefix sum can be performed over the histogram to compute offsets in the update CSR array. Finally, a second pass over the

[1]The back-propagation kernel that takes the *adjoint* $\overline{A}$ (i.e. $\frac{\partial y}{\partial A}$ where y is the loss of the model) and computes $\overline{M}$.

Word	Definition
Featuresize	The number of element in a feature vector (embedding table element).
Accesses	The number of feature vectors accessed and reduced (via a binary operator like elementwise addition) to compute a single output vector.
Batchsize	The number of independent lookup operations grouped together for computational benefit.
Ensemble	Collection of embedding tables, each of which is accessed via independent lookup operations. In the case of DLRM, the outputs of an ensemble lookup are concatenated together to form a large two-dimensional array. This post-op concatenation can be fused with the ensemble lookup.

TABLE 5.1. Parameters that define a particular instance of the embedding table lookup and gradient descent update problem.

indices in M is used in conjunction with both the histogram and the previous prefix sum to populate the entries of the CSR array[2].

The computation of the adjoint $\overline{M}$ is followed by the application of the gradient descent algorithm (not shown in Figure 5.2) where the new embedding table M' is computed via $M' = M - \alpha\overline{M}$ for a learning rate $\alpha \in \mathcal{R}$ [**58**]. While other optimizers like momentum or ADAM can be used, we focus on simple SGD in this work.

Table 5.1 presents a taxonomy of parameters. Figure 5.2 illustrates the case where three rows in the embedding table are summed to produce a single output row ($accesses = 3$) and three such independent operations are performed ($batchsize = 3$). It could be the case that $accesses = 1$, in which case each independent lookup operation is essentially a *memcpy*. Furthermore, a particular instance of a DLRM model may use an *ensemble* of embedding tables, in which case each embedding table experiences its own lookup and update operations.

5.2.2. Embedding Table API. We seek to apply our framework ideas to embedding tables lookup and updates. To do this, we separate the algorithm implementation from the embedding table data structure implementation using the small, well defined API shown in Table 5.2. The function `featuresize` simply return the number of elements (e.g. `Float32`, `Float16`, etc.) within each vector. This is implemented such that it may be known at compile time, enabling code generation customized for a particular featuresize. The function `rowpointer` returns a pointer to the first element

[2]The curious reader can see the implementation of this reindexing procedure at the following link: **https://github.com/darchr/EmbeddingTables.jl/blob/269f0b3c5b295af21aef2160c60f0efe2ad635ea/src/utils.jl#L87-L192**

Function	Description
`eltype(A)`	Return the element type (e.g. `Float32`, `Float16`) of embedding table A.
`featuresize(A)`	Return the number of elements in each row of embedding table A. **Note:** This may be known at compile time.
`rowpointer(A, i, [context])`	Return a pointer to row i of embedding table A. May additionally take a `context` that denote lookup or updating context.
`nvectors(A)`	Return the number of vectors stored in embedding table A.
`example(A)`	Return an instance of some backing array in embedding table A that may be passed to `similar`.

TABLE 5.2. Minimal API required for our embedding table library implementation.

```
 1  function lookup!(dst::AbstractArray, src::AbstractEmbeddingTable, indices)
 2      @inbounds for (dst_row, src_row) in enumerate(indices)
 3          src_ptr = rowpointer(src, src_row, Forward())
 4          dst_ptr = rowpointer(dst, dst_row, Forward())
 5          for i in OneTo(featuresize(src))
 6              x = unsafe_load(src_ptr, i)
 7              unsafe_store!(dst_ptr, x, i)
 8          end
 9      end
10      return dst
11  end
12
13  function lookup(src::AbstractEmbeddingTable, indices::AbstractVector{T}) where {T <: Integer}
14      # Range check
15      @assert all(>(zero(T)), indices)
16      @assert all(<=(nvectors(src)), indices)
17      # Allocate destination and perform lookup
18      dst = similar(example(src), eltype(src), (featuresize(src), length(indices)))
19      return lookup!(dst, src, indices)
20  end
```

LISTING 5.1. Sample embedding table lookup implementation. The optional context `Forward` may be used by the embedding table implementation to specialze for this operation. The function `lookup!` is implemented such that the destination array may be an array view, allowing for fusion with a post-op concatenation.

of the requested row. We provide an optional `context` argument that includes types like `Forward` and `Update`, allowing embedding table implementations to specialize their pointer return strategy. The final two functions are straightforward, `nvectors` returns the number of vectors in a table, allowing for bounds checking, and `example` provides a mechanism to use custom array types for the embedding table and to have these custom arrays propagated to the destination.

Example usage of this API to implement a non-reducing embedding table lookup ($accesses = 1$) is shown Listing 5.1. The top level function `lookup` (line 13) performs bounds checking, allocates

```
 1      .text
 2      mov r8, qword ptr [rdi + 32]            # Load Length of Index Vector
 3      test    r8, r8                          # Check if Index Vector is Empty
 4      je  L88
 5      mov r9, qword ptr [rdx]                 # Base Pointer of Index Vector
 6      mov rax, qword ptr [rsi]                #
 7      mov rdx, qword ptr [rax]                # Base Pointer of Source Array
 8      mov rsi, qword ptr [rdi]                # Base Pointer of Destination Array
 9      mov rdi, qword ptr [rdi + 24]           # Destination Featuresize
10      add rsi, 32                             # Shift base pointer
11      shl rdi, 2                              # Adjust Featuresize for Float32 elements
12      xor eax, eax
13      nop word ptr cs:[rax + rax]
14  L48:
15      mov rcx, qword ptr [r9 + 8*rax]             # Load lookup index
16      shl rcx, 6                                  # Adjust for static featuresize
17      vmovups ymm0, ymmword ptr [rcx + rdx - 64]  # Load data into AVX Registers
18      vmovups ymm1, ymmword ptr [rcx + rdx - 32]
19      vmovups ymmword ptr [rsi - 32], ymm0        # Store data to destination
20      vmovups ymmword ptr [rsi], ymm1
21      add rsi, rdi                                # Increment dest base pointer
22      inc rax                                     # Increment Index Vector Index
23      cmp r8, rax                                 # Check if done
24      jne L48
25  L88:
26      vzeroupper
27      ret
28      nop dword ptr [rax]
```

Listing 5.2. Generated x86 assembly code for an embedding table lookup operation. In this case, each vector in the embedding table consists of 16 `Float32` elements. The featuresize (16) is encoded in the type domain, allowing the compiler to specialize on this amount. As a result, the generated code for the memory copy (lines 17-20) is completely unrolled.

the destination array, and then delegates the implementation to `lookup!` (line 1). Now, `lookup!` is implemented such that the destination array `dst` is simply an `AbstractArray`. In particular, this allows `dst` to be a view into a larger array, allowing fusion the post lookup concatenation used in DLRM.

The example code in Listing 5.1 over simplifies the implementation. In practice, we use aggressive loop unrolling and LLVM vectorizer hints to obtain light-weight loop assembly. We also supply sufficient machinery such that compile time optimizations can be applied if the result of `featuresize` is known at compile time. These optimizations include simpler loop level logic (or even complete loop unrolling) since the number of iterations required for a memory copy or vector addition are known at compiler time. Furthermore, if the featuresize is small enough, we can keep all intermediate partial sums within and x86 CPU's vector registers, significantly reducing pressure on the L1 cache. This effect is demonstrated in Listing 5.2, which shows the light-weight x86 assembly for a non-reducing embedding table lookup for a table with a statically-known featuresize of 16 `Float32`. Note that the memory copy operation (lines 17-20) is completely unrolled and implemented in just 4 instructions! These optimizations are applied to both the lookup and pullback operations, with the capability of fusing a model optimizer (like SGD or Adam) directly with the pullback implementation, avoiding the need to allocate $\overline{M}$ entirely.

5.2.3. Parallelization Strategy. In the case of DLRM, it's natural to parallelize lookups across the ensemble of embedding tables. However, just on its own, this parallelism may not fine-grained enough for efficient load balancing. Observe that if our lookup or update operations use data structures like the CSR arrays shown in Figure 5.2, then (1) each entry in the *offsets* vector corresponds to a unique destination and (2) the *offsets* vector can be efficiently partitioned into smaller chunks since its elements are uniform in size. A thread can work on one of these chunks and update the corresponding destination rows lock free. Thus, we supplement the inter-table parallelism with intra-table parallelism to achieve good load balancing across many threads.

5.3. Embedding Table Experiments

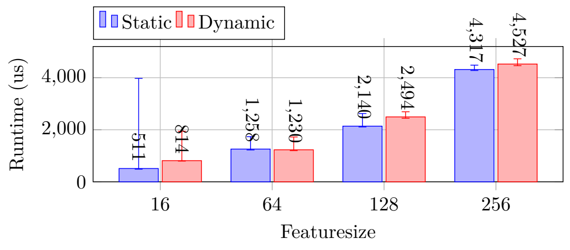
In this section, we will explore the tradeoffs in some of the performance optimizations made in the embedding table library. This information will be used to drive future memory management procedures.

5.3.1. Methodology. A wide number of tests were conducted over the design space described below.

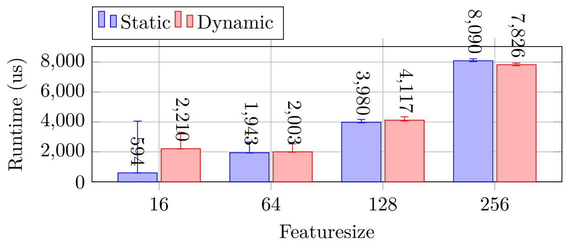
- **Single vs Multiple Threads:** While the full implementation of the lookup and update operations uses all available threads on a socket to maximize performance, running single table lookups on a single thread can help distinguish which parts of the implementation are computationally bound and which are memory bound.
- **Static vs Dynamic Cases:** As mentioned in previous sections, the embedding table library has the capability of storing the featuresize as a compile-time parameter (which will be referred to as the *static* case. The *dynamic* case stores the featuresize as a runtime variable (though this parameter has the same value in both cases). The static case allows

for optimizations not available to a simple implementation of the dynamic case, including complete loop unrolling and storing partial sums in registers.

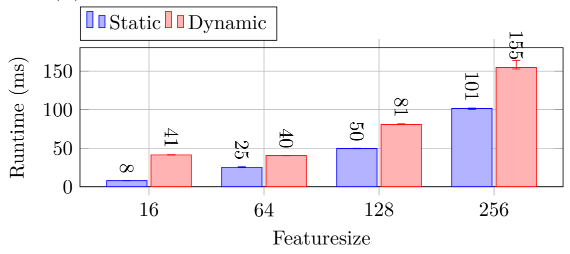
- **Featuresize:** We sweep the featuresize from 16 to 256 (with element type of `Float32`). Smaller feature sizes have smaller random reads (worse for memory bandwidth performance) but lower computation requirements.
- **Number of Accesses:** Non-reducing lookups have congruent read and write bandwidths while reducing lookups with a high number of accesses are more read heavy.
- **Number of Tables:** We vary the number of tables in an ensemble from as low as 10 to as high as 80. Lower numbers of tables can require intra-table partitioning for better load balancing.
- **Table Location:** Placing the tables in DRAM and PM has a considerable impact on performance.
- **Direct vs Indirect Lookup:** Eventually, we would like to manage the location of each individual featurevector. Doing this would require adding a level of indirection to each vector access. That is, one memory access to retrieve the pointer to the vector and one more memory access to retrieve the vector. This is opposed to the direct lookup method that stores all vectors in a contiguous region of memory where vector access is performed through a base plus offset calculation.
- **Standard vs Non-Temporal Stores (Update only):** When conducting the final write of an embedding table update, the library provides the ability to use either standard or non-temporal stores. Note that in this case, non-temporal stores *do not* elide the read-for-ownership for the corresponding cache lines because these lines were already read as part of the application of gradient descent. However, using non-temporal stores *can* provide some measure of grouping together write to adjacent memory locations.
- **Number of Worker Threads (Update only):** Based on the data presented in Figure 2.2b, the write performance of Optane PM may degrade when using a large number of threads. To that end, we also conduct gradient descent update operations with fewer threads.



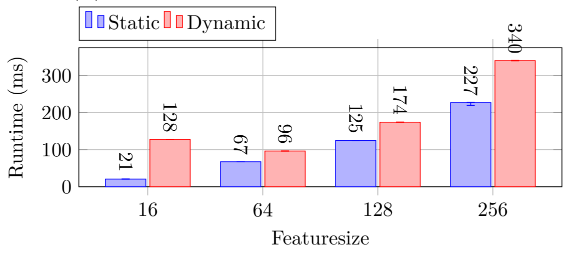
(a) Non-reducing lookup using DRAM.



(b) Non-reducing lookup using PM.



(c) Reducing lookup (40 accesses) using DRAM.



(d) Reducing lookup (40 accesses) using PM.

FIGURE 5.3. Comparing embedding table lookup performance with a single thread, single-precision element types between statically sized and dynamic kernel sizes. All runs used a batchsize of 16384 and $nvectors = 10,000,000$.

All experiments were conducted on a single NUMA node. Single-threaded experiments were pinned to a single core within that NUMA node to avoid any core-migration by the operating system. The experiments consisted of running the kernel of interest multiple times until 20-seconds of wall-clock time had elapsed, the execution time for each invocation was logged. For each invocation of the kernel, new lookup/update indices where generated randomly from a uniform distribution. Execution time for the gradient descent update kernels includes the time for reindexing. In addition to execution time, hardware performance counters for DRAM and PM read and write traffic were also collected.

All experiments used a large batchsize of 16384. This is high enough to reliably test the memory subsystem and batch sizes this large can be seen in practical DLRM training instances. For exmple, submissions to MLPerf training [**63**] often use batchsizes in the high 10s to low 100s of thousands. Embedding tables were sized to occupy a memory footprint between 1 GiB and 80 GiB to minimize the effect of the L3 cache. The `Float32` element type was also used for all experiments.

5.3.2. Results. In this section, we present relevant and interesting results from the large number of experiments conducted.

5.3.2.1. *Static and Dynamic Featuresize.* Figure 5.3 shows runtime differences between static and dynamic featuresizes for both non-reducing and reducing ($accesses = 40$) single embedding table lookups. It further shows performance for both DRAM and Optane PM. In the non-reducing case, the static case only speeds up the case where the vector featuresize is 16. This is because the dynamic case uses loop unrolling that is too aggressive and hence slower fallback code is used. For the larger featuresizes, the unrolled innerloop is executed in the dynamic case, bringing the performance on-par with static case.

The story is different for the reducing experiments where the dynamic case is consistently 50% slower than the static case. In the static case, intermediate partial sums are kept inside the AVX-512 registers while in the dynamic case the partial sums must be spilled into the L1 cache. This suggests an strategy for the dynamic case is to tile the lookup aggregation into statically sized chunks instead of operating on the entire feature vector directly. However, while this improves computational performance, it (1) loses the bandwidth efficiency of featuring the entire featurevector as a contiguous chunk and (2) increases the kernel code footprint to include all possible corner cases.

Finally, the performance of PM in these applications is on the order of 2× slower than DRAM demonstrating that even for a single thread, memory location matters.

This demonstrates that kernel implementation matters and knowledge of the underlying hardware is key to achieving high performance for these types of workloads.

5.3.3. Efficiency of Parallel Lookup Implementation. To check the performance of the parallel ensemble lookup, hardware performance counters were used monitor DRAM and PM bandwidths. The results for one such benchmark are given in Table 5.3. When the tables are located in DRAM, we achieve close to 100 GB/s of read bandwidth. This is close to the theoretical bandwidth of 110 GB/s achieved in Figure 3.3, especially considering that lookups are largely random instead of sequential.

The PM bandwidth achieved during ensemble lookup is somewhat less than what can be achieved ideally (see Figure 2.2a) but still seems reasonable considering the mix of DRAM reads and write that must occur on the same physical DDR bus.

Featuresize	Table Location	DRAM Read	DRAM Write	PM Read	PM Write
16	DRAM	97.2	2.45	0	0
64	DRAM	98.3	2.29	0	0
256	DRAM	103.6	1.57	0	0
16	PM	0.78	0.28	10.4	0
64	PM	0.36	0.47	18.2	0
256	PM	0.12	0.39	25.0	0

Table 5.3. DRAM and PM bandwidth (given in GB/s) for reducing ensemble lookups with 80 tables (1 million columns each), 40 accesses per output, batchsize 16384, and 28 worker threads.

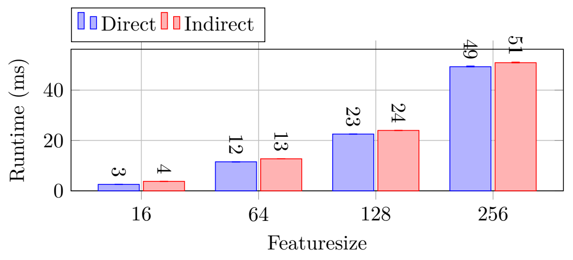
From this, we can conclude that the lookup implementation is reasonably performant, achieving close to the theoretical bandwidth of the platform.

5.3.4. Adding Indirection to Lookup Operations. Figure 5.4 shows the effect that adding an extra level of indirection to each vector access has on the performance of an ensemble lookup. An extra level of pointer chasing causes a slight slowdown when embedding tables are in DRAM and roughly performance parity when the tables are in PM. In this bandwidth constrained environment with a large number of threads, the overhead introduced by an extra level of pointer chasing is negligible. The largest performance loss occurs for a DRAM based lookup with 40 accesses and a featuresize of 16 where the amount of data moved for each vector access is relatively small.

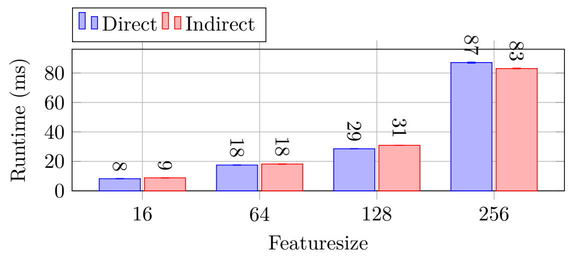
Therefore, we should be able to add a level of indirection, allowing individual feature vectors to be located in either DRAM or PM, without a large sacrifice in performance.

5.3.5. SGD Update Performance - Worker Threads and Nontemporal Stores. Figure 5.5 shows an example ensemble gradient update performance broken down between DRAM and PM, number of worker threads, and usage of standard versus non-temporal stores. The behavior of DRAM (Figures 5.5a and 5.5b) is straightforward - performance increases with the number of threads with little performance difference between standard and nontemporal stores during the update phase.

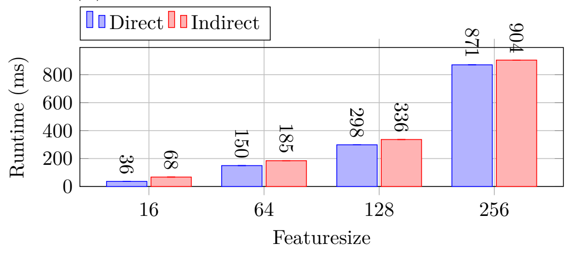
Persistent memory (Figures 5.5c and 5.5d) exhibits more nuanced behavior. Because the write bandwidth to PM is much lower, reindexing time is less of a bottleneck than it is for DRAM. Furthermore, non-temporal stores tend to perform significantly better, especially for larger feature



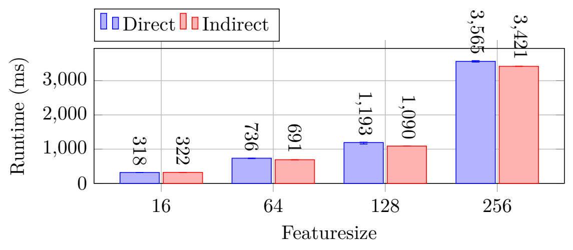
(a) Tables in DRAM with 1 access.



(b) Tables in PM with 1 access.



(c) Tables in DRAM with 40 accesses.



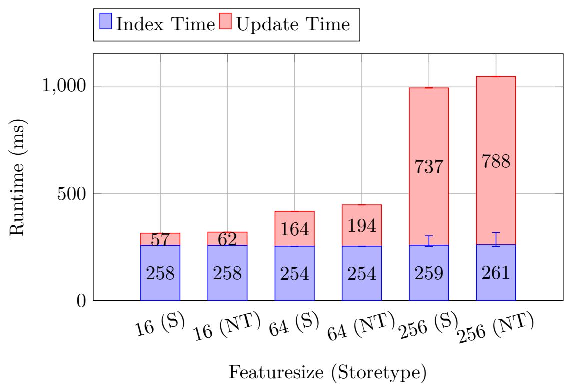
(d) Tables in PM with 40 accesses.

FIGURE 5.4. Ensemble lookup performance when adding an extra level of indirection for each vector access. 80 independent tables were used with 1 million vectors each along with 28 worker threads and a batchsize of 16384.

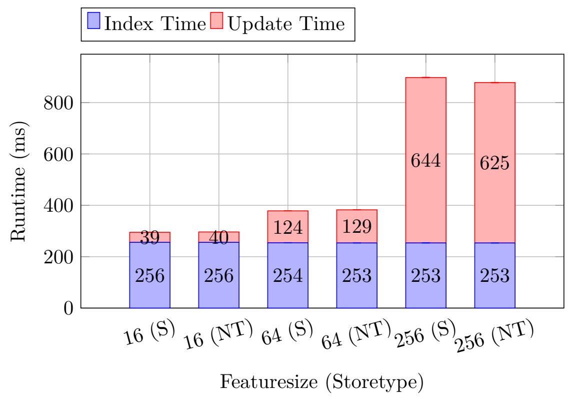
sizes. As mentioned previously, this is not because the RFO for the corresponding data is elided since this data is already cached from a previous read. Instead, this is likely because non-temporal stores evict the corresponding cachelines from the cache. This causes the writes to appear at the memory controller essentially as a group allowing for write-combining within the Optane memory controller (recall that this generation of Optane DIMMs have a 256 B access granularity). Without non-temporal stores, the corresponding cache lines only arrive at the memory controller when evicted from the L3 cache, leading to lower spatial locality.

For these experiments, the time taken by the reindexing procedure is mostly constant and takes a large fraction of the overall execution time when the embedding tables are in DRAM. This is largely because the reindexing procedure is largely targeted for situations where the number of unique indices accessed is relatively small compared to the number of vectors in the table. A choice of data structures and reindexing operation targeted more specifically at this “high density” situation may reduce the this time.

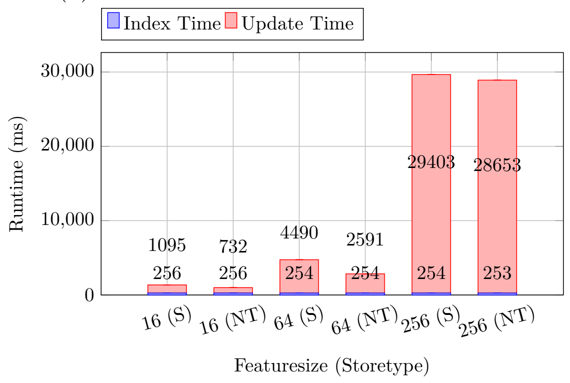
5.3.6. Effect of Hardware Prefetchers. Modern CPUs use hardware prefetchers which can detect patterns in load instructions and preemptively fetch cache lines consistent with those



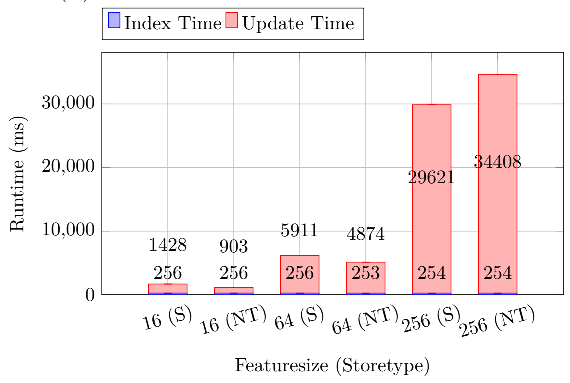
(a) Tables in DRAM with 12 threads.



(b) Tables in DRAM with 28 threads.



(c) Tables in PM with 12 threads.



(d) Tables in PM with 28 threads.

FIGURE 5.5. Embedding table SGD application performance comparing the use of nontemporal and standard stores. The type of store is indicated by (S) for standard and (NT) for nontemporal. 40 independant tables were used with 1 million vectors each, 40 tables accesses per output, batchsize 16384.

patterns. One in particular is called the "streaming prefetcher" [**90**], which can detect patterns of loads to sequential memory addresses. This can have a detrimental impact on embedding table performance due to over fetching. To understand this, we first need to compute the expected amount of data moved for embedding table lookup and update operations.

For lookups, this can be computed as follows:

$$table\_accessed = num\_tables * batchsize * accesses * featuresize * sizeof(Float32) \tag{5.1}$$

where *table_accessed* is the amount of bytes read from an ensemble of embedding tables,

$$index\_accessed = num\_tables * batchsize * accesses * sizeof(UInt32) \tag{5.2}$$

where *index_accessed* is the number of bytes read from the index arrays used to access the embedding tables, and

$$destination\_accessed = num\_tables * batchsize * featuresize * sizeof(Float32) \tag{5.3}$$

is the number of bytes written to the output arrays.

The update process is a little harder to estimate because of the reindexing procedure. However, we can still estimate the amount of traffic generated to the embedding table. This traffic depends on the number of *unique* rows accessed during a lookup. For a uniform index distribution, the expected value for unique rows is

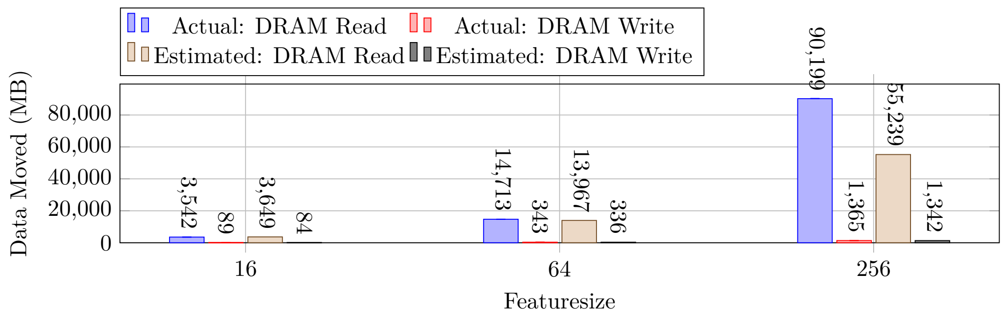
$$unique\_rows \approx n\left(1 - \left(\frac{n-1}{n}\right)^{k}\right) \tag{5.4}$$

where n is the number of rows in each table and k is the total number of independent accesses to the table. In the context of embedding table operations, $k = batchsize * accesses$. From this, we have

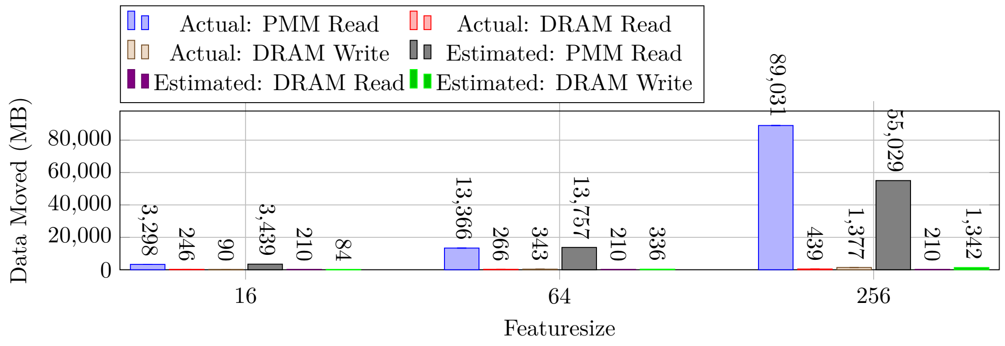
$$bytes\_accessed \approx num\_tables * unique\_rows * featuresize * sizeof(Float32) \tag{5.5}$$

Figure 5.6 compares the estimated amount of data moved with the actual data moved for embedding table lookup and updates as measured by hardware performance counters. Figures 5.6a and 5.6b show ensemble lookups with the embedding tables in DRAM and PM respectively. In general, the estimates align well with the measured movement until featuresize 256, where the amount of data accessed from the table is significantly higher than expected. This is because a featuresize of 256 (corresponding to a contiguous access of 1024 bytes) is long enough to trigger the streaming prefetcher into fetching more data than necessary. Because ensemble lookups are bandwidth limited, this degrades overall throughput. We observe a similar phenomenon in Figure 5.6c, which just shows the memory traffic to an embedding table ensemble during a gradient descent update operation. Again, at a featuresize of 256, the amount of data read from PM is nearly double the expected amount.

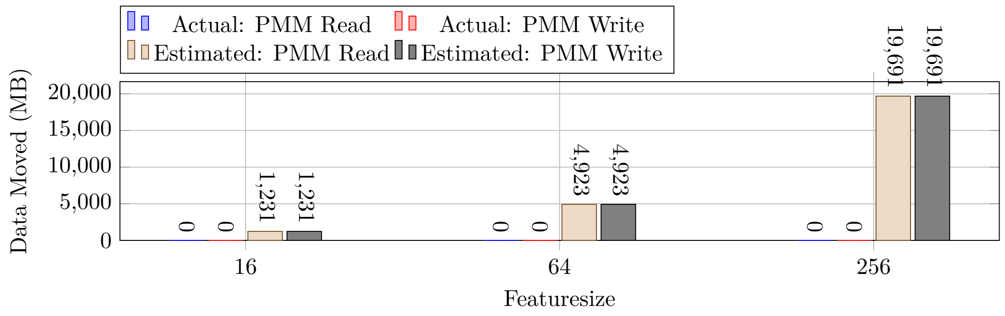
With the streaming prefetcher enabled, this suggests again that it might be beneficial to tile lookups/updates for large featuresizes into multiple smaller operations with a sub-featuresize small



(a) Lookup - table in DRAM.



(b) Lookup - table in PM. In this case, DRAM read traffic is the result of accessing the index arrays.



(c) Update - data in PM. This figure just shows the traffic to the embedding table. PM reads for featuresize 256 are significantly higher than expected.

FIGURE 5.6. Comparison of estimated and actual data moved for embedding table lookup and update operations as measured by hardware performance counters. For small featuresizes, the estimated and actual values are close. However, for a featuresize of 256, the amount of data read is significantly higher than anticipated because hardware prefetchers are detecting the streaming pattern and over-fetching data. This phenomenon disappears when the *streaming prefetcher* is disabled.

enough to evade detection by the prefetcher. This would lose some streaming locality in the memory accesses, but would synergize well with our previous observation of keeping partial sums in the AVX registers as much as possible. The phenomenon of over-fetching disappears when the streaming prefetcher is disabled in the BIOS. However, disabling the streaming prefetcher has implications for computational based workloads. To that end, we decide to leave the prefetcher enabled and take the performance regression for larger featuresizes, recognizing that there may be future optimization opportunities.

5.3.7. Discussion. There are a number of conclusions that can be drawn from these embedding table experiments. First, placing the tables in PM results in lower performing lookup and update operations than DRAM. This is expected within the context of what has discussed earlier in this dissertation. Further, this highlights the need to perform some kind of heterogeneous memory management to get the capacity advantage of PM without paying the full performance price.

Second, higher performance implementations of embedding table operations requires cooperation with and understanding of the underlying hardware and the best implementation can change depending on the particular operation. For example, the use of nontemporal stores for update operations is beneficial for performance when embedding tables are in PM, but makes little difference when DRAM is used.

Finally, in the context of multithreaded ensemble lookups and updates, an extra level of indirection can be tolerated without much of a performance penalty. This is the main idea behind our idea of memory management for these tables which will be presented in the next section. Adding this indirection allows individual vectors to be stored in either PM or DRAM. With careful selection, we should be able to move frequently accessed vectors into DRAM while leaving infrequently accessed ones in PM, providing most of the performance of an all DRAM with the capacity advantage of PM.

5.4. Software Caches for Gigascale Embedding Tables

In this section, we discuss how to apply the framework of heterogeneous memory management to embedding table lookups and updates into an approach called *CachedEmbeddings*. Key aspects to keep in mind are that (1) access to each embedding table is performed on the granularity of feature vectors, (2) there is no reason to expect accesses to exhibit spatial locality, and (3) accesses *may* exhibit temporal locality. The key insight of *CachedEmbeddings* is to add an extra level of indirection to each feature vector access, allowing individual feature vectors to be placed in either

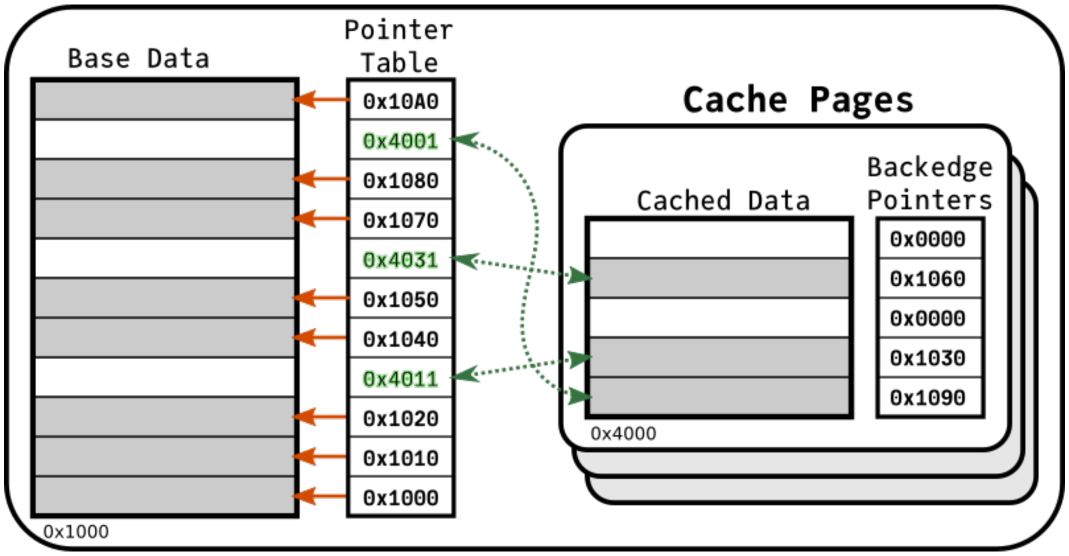


FIGURE 5.7. Overview of a *CachedArrays* embedding table. Base data lives in PM, (with a base address of `0x1000` as an example). In this example, each feature vector occupies 16 bytes. A pointer table tracks the actual location of each vector with the least significant bit indicating whether it's cached. Upon a lookup access, vectors are moved into cache pages. Each page contains backedges for each entry, which indicates whether the corresponding slot is filled and if so, the vectors original location.

PM or DRAM. From Section 5.3.4, we observe that adding this extra level of indirection doesn't drastically harm performance in highly-parallel scenarios. Here, we define a software cache to exploit this.

Figure 5.7 shows an overview of our approach. In the vocabulary of our management framework, the abstract feature vectors (rows) of the embedding table are the "objects" with the actual pointers to where each vector resides being the "regions". Base data for the embedding table is located in PM (beginning at address `0x1000` in the example). Each embedding table maintains a cache in DRAM that vectors can be migrated to. Internally, the embedding table maintains a vector of pointers, one for each row, pointing to where the primary region for that row is. Since embedding table rows are relatively large ($> 64\,B$), these pointers have unused lower order bits. We use the least significant bit (LSB) to encode whether the corresponding row is in the base data or in a cache page. The second LSB is used as a lock-bit. A thread wanting to move a row uses an atomic compare-and-swap to gain ownership of the row. If ownership is acquired, then the thread is free to move the row into the cache and then unlock the row.

To support multithreaded access, the cache is composed of multiple cache pages. In order to allocate space for a feature vector, each cache page maintains a bump pointer. Upon an allocation, the bump pointer is atomically incremented with the old value of the pointer serving as the location for allocation. A page becomes full when this bump pointer reaches the end of the memory region allocated for the page. If the most recent cache page is full, then the thread must acquire a lock for the table in order to allocate new cache page. The cache has a configurable maximum size, beyond which no more feature vectors can be migrated until the cache is flushed. Each cache page also maintains a vector of backedge pointers to each cached row's original location (or null if the slot is empty) to facilitate this flushing.

Because bump pointers are used to allocate space for feature vectors, the cache is flushed one page at a time. If the cache page is entirely clean (in the case that only lookups were performed with no update operations), flushing a cache page simply involves updating the *pointer table* back to each vector's original location and then deleting the cache page. If the vectors are dirty (e.g. the table was used during training) then the vectors within the cache page must also be written back to their original location.

The size of the cache is determined by two parameters:

- **cachelower:** Soft lower bound for the size of the cache. When the cache is flushed, pages will be sequentially flushed until the size of the cache is less than `cachelower`.
- **cacheslack** Flexible space to allow the cache to grow. New vectors can be cached until the total size of the cache exceeds `cachelower + cacheslack`.

Thus, the size of the DRAM cache for each table can fluctuate between `cachelower` and `cachelower + cacheslack`. Note that the size of the cache can also be *lower* than `cachelower` at the very beginning of program execution when no vectors have been cached.

Table 5.4 outline the API for a CachedEmbeddingTable. The functions `access_and_cache` and `access` provides methods for retrieving feature vectors while optionally migrating vectors into the table's DRAM cache. Setters `set_cachelower` and `set_cacheslack` are used to modify their corresponding cache size parameter variables. Finally, `flush_clean` and `flush_dirty` provide methods for reducing the size of the cache to enable future vector accesses to be cached.

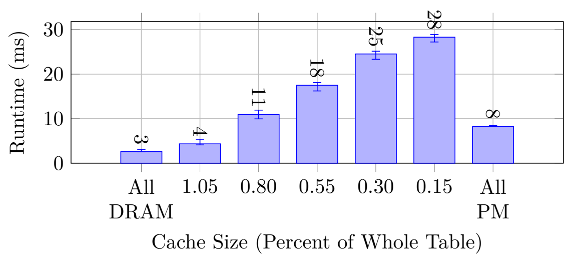
Operation	Description
`access_and_cache`	Get the pointer for the requested feature vector, caching it in DRAM if (1) the cache is not full, (2) the vector is not already cached, and (3) ownership of the row is acquired. Internally, this is connected to the `rowpointer` function introduced in Table 5.2 and connected to the `Forward` access context.
`access`	Get the pointer for the requested feature vector without caching. This function is connected to the `rowpointer` function for all other access contexts besides `Forward`.
`set_cachelower`	Set the `cachelower` variable.
`set_cacheslack`	Set the `cacheslack` variable.
`isfull`	Return `true` if the cache is full. Otherwise, return `false`.
`flush_clean`	Purge the oldest cache pages until the size of the cache is less than `cachelower`. Do not write back data from cache pages to the base array.
`flush_dirty`	Purge the oldest cache pages until the size of the cache is less than `cachelower`. Do write back data from cache pages to the base array.

Table 5.4. API for a CachedEmbeddingTable.

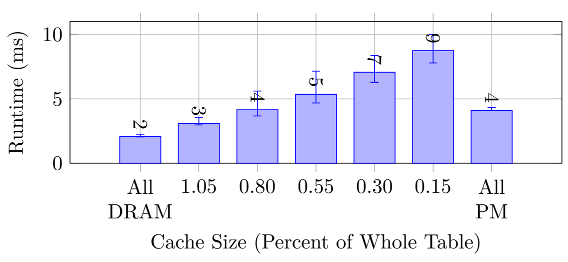
5.4.1. *CachedEmbeddings* Performance. In this section, we perform experiments to determine the performance of the *CachedEmbeddings*.

5.4.1.1. *Methodology.* When comparing the performance of *CachedEmbeddings* to standard embedding tables, we focus on the lookup operation performance. This is because, in the context of DLRM training, feature vectors will be cached in DRAM during the lookup operation and simply accessed during the gradient descent operation. The performance of this update operation and subsequent cache flushing is harder to micro-benchmark for a couple of reasons. First, in the context of DLRM training, we'd expect all embedding tables entries accessed during the update phase to already be cached. Second, the frequency of a flush operation is dependent on the input index distribution and thus doesn't necessarily occur on every training iteration. Consequently, we will examine update performance when we study then end-to-end performance of DLRM with *CachedEmbeddings*.

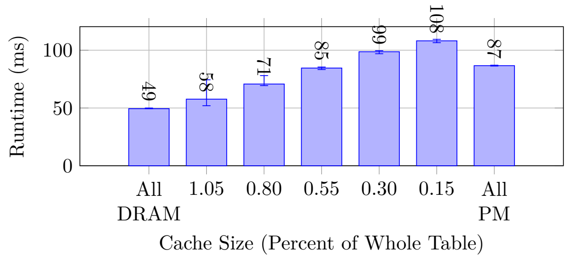
For our benchmarks, we want to target conditions where a mix of DRAM and PM makes sense (i.e., the total memory footprint is high). To that end, we investigate ensemble lookups with 80 tables and 28 threads with featuresizes of 16 and 256 and accesses of 1 and 40. Furthermore, each table consisted of 1 million vectors and a batch size of 16384 was used. To investigate the effects



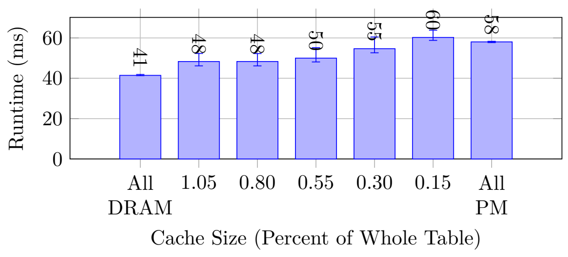
(a) Featuresize 16 - Uniform Distribution



(b) Featuresize 16 - Zipf ($\alpha = 1$) Distribution



(c) Featuresize 256 - Uniform Distribution



(d) Featuresize 256 - Zipf ($\alpha = 1$) Distribution

FIGURE 5.8. Comparison of *CachedEmbeddings* with standard embedding tables located in DRAM or PM for *nonreducing* lookups. Runs were conducted with 80 embedding tables and 28 worker threads.

of cache size, we set `cacheslack` to be 5% and `cachelower` to 10%, 25%, 50%, 75% and 100% of each table's total memory footprint.

To investigate the effects of temporal locality, the lookup indices for each table are drawn from either a uniform distribution (which has low temporal locality) or a Zipf [**79**] distribution with $\alpha = 1$ (which has high temporal locality). In order to avoid spatial locality introduced by the Zipf distribution, the index sampling is followed by a maximum length linear feedback shift register (LFSR) using a different seed for each table.

For comparison points, the same experiments were run for standard embedding table with either all data stored in DRAM or PM and no indirection in the lookup accesses.

As before, each lookup operation is invoked multiple time with different indices until the total benchmark runtime exceeds 20 seconds. For the experiments conducted using *CachedEmbeddings*, the `flush_clean` operation is run after each invocation.

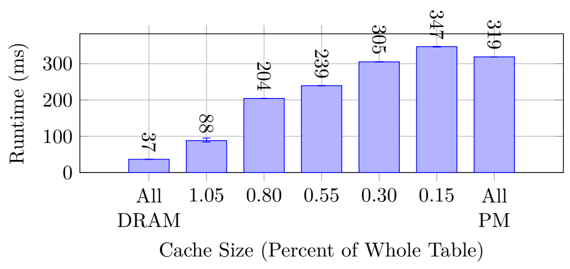
5.4.1.2. *Results.* Figure 5.8 shows the results for a non-reducing embedding table ensemble lookup. The left-most and right-most bars in each figure show the performance of a standard embedding table with all DRAM and PM respectively. In between is shown the performance of a

CachedEmbeddings, with the label giving the sum of `cachelower` and `cacheslack` as a percent of the ensemble's total memory footprint.

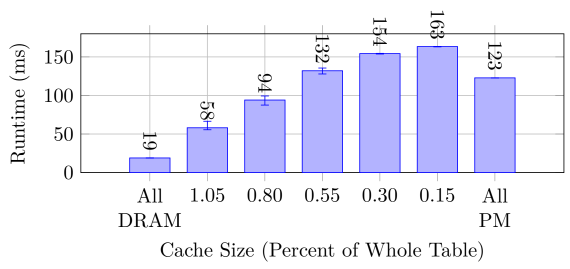
Initially, this does not look too great for *CachedEmbeddings*. For a featuresize of 16 (Figure 5.8a and 5.8b), the overhead of cache management overheads dominates resulting in significant slowdown over the all PM simple table. Even with the larger featuresize of 256, *CachedEmbeddings* requires a fairly large cache size to outperform the all PM standard table.

There are a number of reasons for this. First, non-reducing lookups are essentially a memory copy from either DRAM to DRAM or PM to DRAM. This higher DRAM write traffic can, to some extent, help mitigate the lower read bandwidth of PM which we can see with the 2× lower performance of the PM based simple tables than the DRAM based ones for the uniform distribution. Second, because `flush_noclean` is called after every invocation and only at most 16384 are accessed on each lookup (around 1.6% of the embedding table) the table never reaches the state where the cache is full (recall that `cacheslack` was set to 5% of the overall table size). This means that the *CachedEmbeddings* table is always doing extra work and cannot necessarily take advantage of preexisting cached vectors.

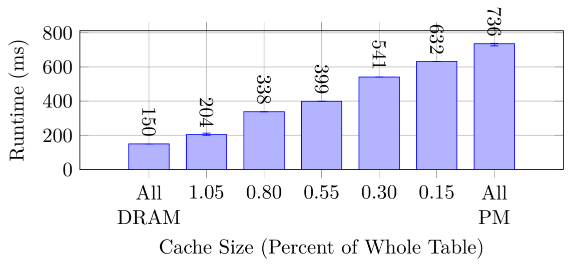
Figure 5.9, on the other hand, provides some hope for the situation. This figure shows the performance of *CachedEmbeddings* for reducing lookups (with $accesses = 40$). Again, the smaller feature sizes yield poorer performance advantages (or even performance regressions at smaller cache sizes) because the time spent moving data around is so low enough that the extra steps required by *CachedEmbeddings* can dominate. However, for larger feature sizes like 64 and 256, the performance of *CachedEmbeddings* nearly interpolates linearly between the performance of all DRAM and all PM. This is because with a batchsize of 16384 and 40 accesses per batch, a large portion of each embedding table is accessed on each lookup operation, resulting in the each embedding table's cache staying "full" for a large portion of the lookup operation. When full, the extra level of indirection for the embedding tables is amortized by the large number of worker threads, providing a performance benefit over all PM when an accessed vector is in DRAM with little overhead when it is not. This effect is magnified with the Zipf distribution which yields a very high DRAM hit rate with only a modest cache size.



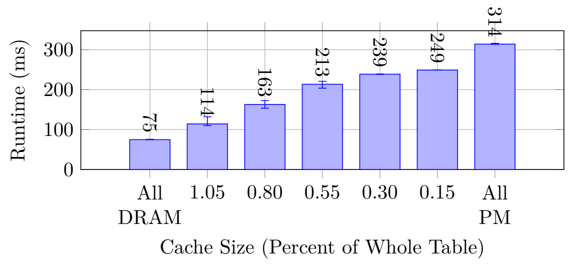
(a) Featuresize 16 - Uniform Distribution



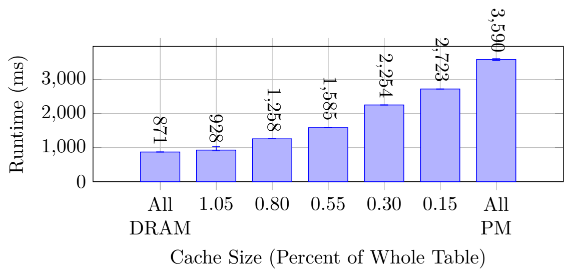
(b) Featuresize 16 - Zipf ($\alpha = 1$) Distribution



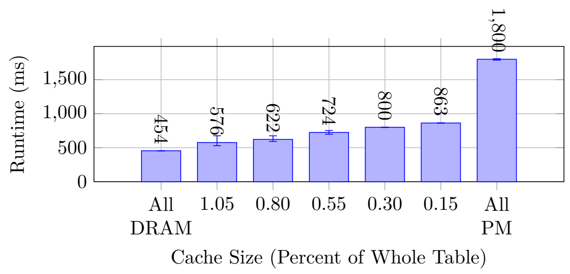
(c) Featuresize 64 - Uniform Distribution



(d) Featuresize 64 - Zipf ($\alpha = 1$) Distribution



(e) Featuresize 256 - Uniform Distribution



(f) Featuresize 256 - Zipf ($\alpha = 1$) Distribution

FIGURE 5.9. Comparison of *CachedEmbeddings* with standard embedding tables located in DRAM or PM for *reducing* lookups with 40 Runs were conducted with 80 embedding tables and 28 worker threads using the preallocation strategy.

5.4.2. Discussion. As we have seen, there are several regimes where this approach of fine-grained heterogeneous memory management can be effective. When the hit rate into the managed DRAM cache is sufficiently high (in the case of the Zipf index distribution) and the feature size is large enough to amortize the overhead of adding indirection to vector access, then *CachedEmbeddings* can outperform all PM with a relatively small amount of DRAM. Even in cases where the hit rate is not particularly high (the case of the uniform index distribution), *CachedEmbeddings* can still achieve a level of performance between all DRAM and all PM provided the cache becomes full and the amount extra work involved on each access decreases. At this operating point, each vector access just adds a level of indirection, sometimes hitting in DRAM and sometimes hitting in PM.

Those accesses to DRAM are accelerated while those to PM have little penalty over the all PM case.

This suggests another use strategy for *CachedEmbeddings* called the *static* approach. If the input distribution is known to have little locality *or* if hot entries in the distribution are known *a priori*, than an appropriate subset of the table can be preemptively moved to DRAM (using `access_and_cache`) until the table's cache is full. At this point, further accesses will only fetch and not move feature vectors. This approach will not respond dynamically to changes in the input distribution, but as we pointed out, may be appropriate is some situations.

Finally, we can discuss using *CachedEmbeddings* in the context of DLRM training. By fetching all accessed feature vectors during the lookup phase of a training iteration, we are guaranteed that all these vectors will be in the DRAM cache during the gradient descent update phase. Depending on the index distribution, this has the potential to accelerate that phase. Whether the overhead of cache management (i.e. `flush_dirty`) are worth it, though, will be investigated over the next couple of sections.

5.5. DLRM Implementation

In order to investigate the performance of CachedEmbeddings in the context of end-to-end DLRM training, we first need implement the rest of the DLRM model. We've already discussed the implementation of embedding table lookup and gradient descent update. In this section, we describe how we implemented the rest of the model and compare its performance a state-of-the-art CPU implementation in PyTorch.

5.5.1. Dense Computation. The rest of our model consisted of MLP layers made up of oneDNN kernels (facilitated by our Julia wrapper library), custom written interaction layers, and usage of `CachedArrays` for memory allocation. The interaction layer is implemented in pure Julia. Note that this shows the flexibility of our approach to implementing DLRM. When applicable, oneDNN can be used to accelerate the operations that are implemented by that library. However, when functionality is needed that is not provided by that library, it can be easily and performantly implemented in a way that inter-operates nicely.

	Small Mode	**Large Model**
Featuresize	16	128
Num Embeddings Tables	26	26
Embedding Table Sizes	min = 3, max = $8.9e6$, $\mu \approx 1.2e6$, $\sigma = 2.6e6$	
Bottom MLP	512-256-64-16	512-256-128
Top MLP	512-256-1	1024-1024-512-256-1
Batchsize	8192	32768

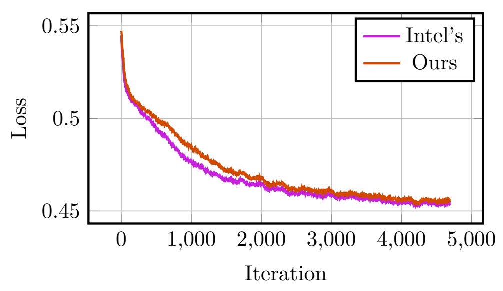
TABLE 5.5. Model hyperparameters used for DLRM PyTorch comparison.

5.5.2. Comparison with Optimized PyTorch (DRAM Only). To verify our model performance, we compared our DLRM implementation Intel's optimized PyTorch [**52**] submission to MLPerf [**63**]. This reference model using custom PyTorch extensions to enable BFloat16 for high performance dense network computations. We were able to acquire temporary access to an Intel Cooperlake server, a generation equipped with vector instructions for BFloat16 based dot products. Since our implementation is build on top of oneDNN (which supports the BFloat16 datatype), we incorporated the BFloat16 data type into our model as well.

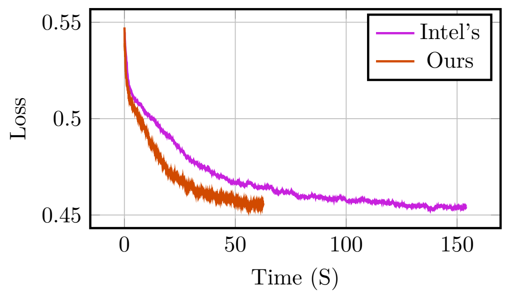
5.5.2.1. *Methodology.* We used two models for comparison, a small model used as Facebook's official DLRM sample model and the model used in MLPerf 2019 training [**63**]. The hyper parameters for these tables is shown in Table 5.5. The optimized PyTorch implementation used *split SGD* [**52**] for their BFloat16 weights. With this optimizer, MLP and embedding table weights are kept in BFloat16, and each weight array is associated with a similar sized array filled with 16-bit integers. During the weight update phase of training, these BFloat16 variables are concatenated with their respective 16-bit integer in their sibling array to create a full 32-bit float. The gradient update is applied to this 32-bit value, which is the decomposed back into a BFloat16 and 16-bit "mantissa". Using this strategy, the authors keep a full 32 bits of precision for training while using 16 bits of precision for inference. Importantly, this technique *does not* decrease the memory requirement of the embedding tables. Consequently, we implement the split SGD trick for the MLP layers of our implementation, but keep our embedding tables in full Float32.

Training data came from the Kaggle Display Advertising Challenge dataset[3]. Both small and large models were run for a single epoch of training on the dataset, iterating over the data in the same order. Further, both our model and the PyTorch model began with the same initial weights.

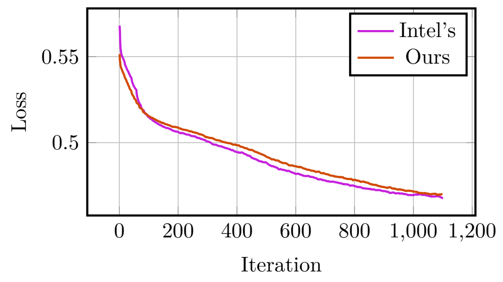
[3]`https://www.kaggle.com/c/criteo-display-ad-challenge/data`



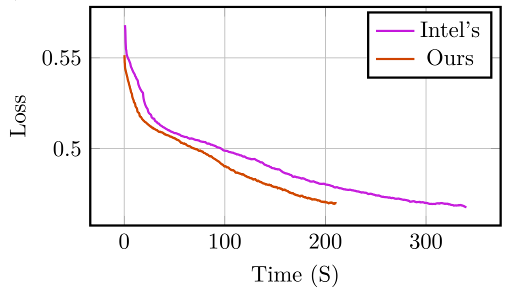
(a) Small model, training loss per iteration.



(b) Small model, training loss through time.



(c) Large model, training loss per iteration.



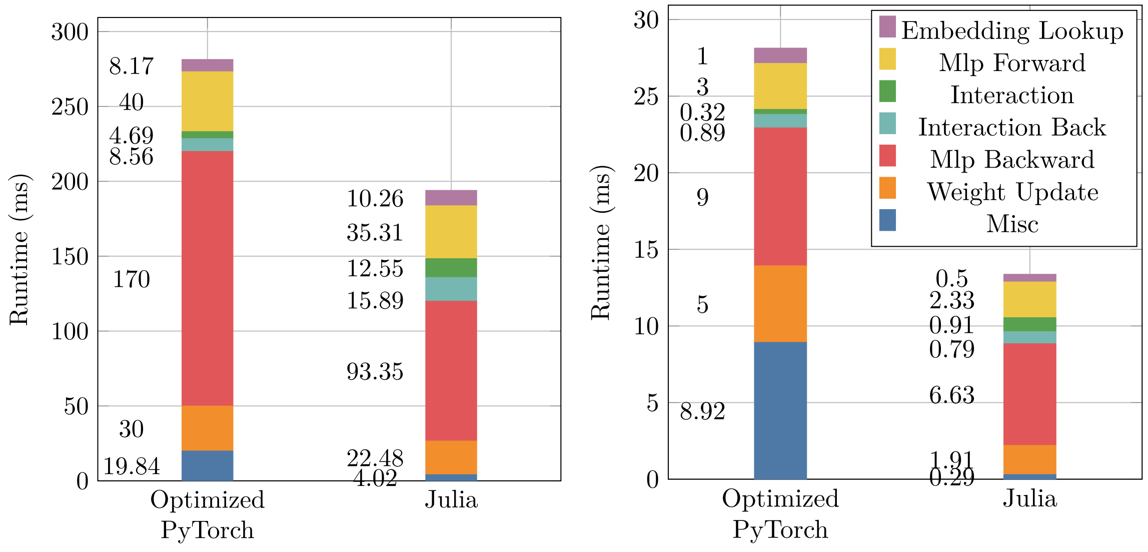
(d) Large model, training loss through time.

FIGURE 5.10. Convergence comparison between the PyTorch optimized DLRM and ours. Our model has a slightly *higher* loss per iteration, but *lower* loss per wall clock time.

5.5.2.2. *Results.* Figure 5.10 shows the loss progression of our model and the optimized PyTorch model for the small and large networks. Figures 5.10a and 5.10c show loss as a function of iteration number while Figures 5.10b and 5.10d show loss as a function of time. We can see that our model has slightly higher (worse) loss per iteration, implying our treatment of BFloat16 is not quite as precise as the PyTorch. However, our model has a significant lead in loss over time because each iteration is processed much more quickly.

Figure 5.11 shows the time breakdown of each iteration for both implementations and models. Our performance benefit comes from three major areas. First, our MLP backward pass is much faster. This is because we are using an up to date version of oneDNN to compute our backward pass kernels while the PyTorch model at the time was using libxsmm[4]. It should be noted that Intel's extensions for PyTorch have since switched to using oneDNN. Second, our implementation has a faster embedding table and weight update through our parallel embedding table update and parallel weight update strategies. Note that even though the wall-clock time for the large network

[4]`https://github.com/hfp/libxsmm`



(a) Large Network (b) Small Network

FIGURE 5.11. Timing breakdown of key layers in our DLRM comparison.

embedding lookup is slightly larger than PyTorch, we're moving twice the amount of data because our tables were kept in `Float32` while PyTorch used `BFloat16`. Finally, our implementation has less miscellaneous overhead, a factor especially apparent for the small network where PyTorch spends a considerable in tensor addition, zeroing, and emptying.

5.5.2.3. *Discussion.* There are two points we would like to make in our brief discussion of these results. First, convergence results are not conclusive as neither model was trained to completion. Nevertheless, this comparison supports our model's correctness and performance. Second, we're a little loose with our loss comparison per iteration compared to PyTorch. There are a large number of sources that can cause such error (e.g., different computation kernels with slightly different numerical results, different treatment of the Float32 to BFloat32 converstions etc.). However, the goal here is to explore the effect of memory optimizations, which don't have an impact on numerical precision. Thus, we're close enough to draw meaningful conclusions.

5.6. End-to-End DLRM Performance of *CachedEmbeddings*

In this section, we investigate the performance of *CachedEmbeddings* for full DLRM training. We investigate several different management schemes built on top of *CachedEmbeddings* and compare their performance with Intel's built-in 2LM hardware managed DRAM cache.

Parameter	Value	Parameter	Value
Number of Tables	64	Rows per Table	6000000
Featuresize	256	Lookups per Output	100
Bottom MLP Length	8	Bottom MLP Width	2048
Top MLP Length	16	Top MLP Width	4096
Batchsize	512		

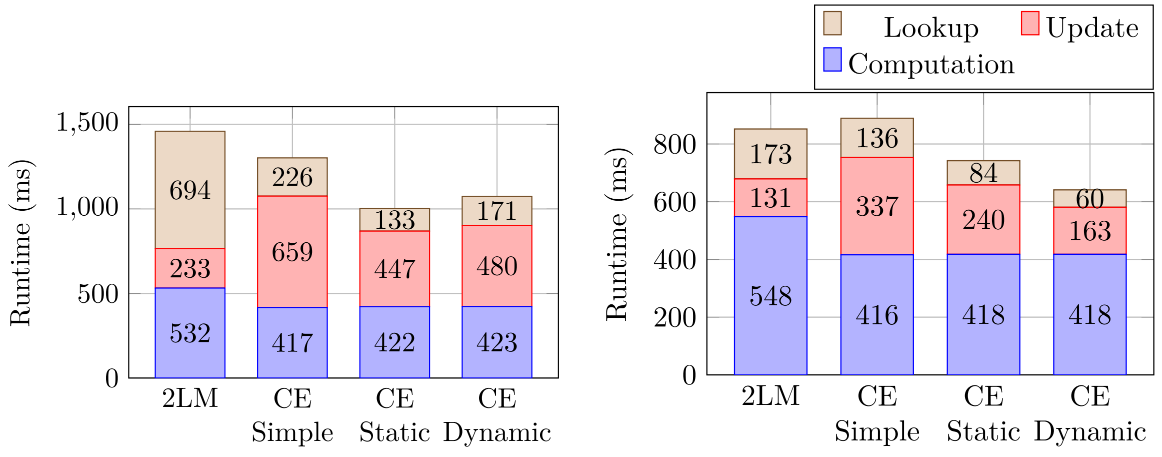
Table 5.6. Parameters for the large DLRM model used for benchmarking.

5.6.1. Policies. We implemented three simple policies on top of *CachedEmbeddings*. The *simple* policy leaves all embedding vectors in PM, using DRAM to store the results of an embedding table lookup and intermediate data for the dense computations. This policy uses a simple embedding table without the level of indirection required for a CachedEmbedding table. The *static* policy allocates a specified amount of memory in DRAM as cache pages, fills these cache pages with random rows, then disables all dynamic row caching. At run time, a row access will either be serviced from DRAM (if one of the rows that was cached ahead of time) or from PM. The *dynamic* policy involves dynamically moves feature vectors into cache pages in DRAM. During lookup of a particular row, the current thread checks if the accessed row is cached and if so directly returns a pointer. If the row is not cached, then the thread attempts to dynamically cache the row using the mechanism described above before returning the pointer. Note that if the row fails to obtain ownership of the row, then a pointer to the base data is given.

Over time, the *dynamic* policy will increase the footprint of the cache pages as more rows are moved into DRAM. In order to compare fairly with *memory mode* (which has access to all of DRAM), we need a per-table cache size small enough to fit in DRAM along side all memory used by the dense computations but large enough to achieve high utilization of the available DRAM. Thus, we set a cache size limit of 2 GiB for each table for a total memory footprint of 128 GiB across the ensemble. Cache pages are sized to be a fraction of this limit and when the limit is reached, the oldest cache page is cleaned up.

If the sparse input distributions are known, then policies can be updated on a per-table basis, (e.g., changing the amount of cache allowed for a table).

5.6.2. Methodology. To test CachedEmbeddings, we used a very large DLRM with the hyper parameters shown in Table 5.6. This model has large and deep MLPs and a memory footprint of



(a) Uniform.

(b) Zipf ($\alpha = 1.0$)

FIGURE 5.12. Performance with different sparse input distributions. Operations “Lookup” and “Update” refer to embedding table lookup and update respectively. All other operations are grouped into “Computation”. Abbreviation “CE” stands for “CachedEmbeddings”.

around 393 GB for its embedding tables. For this large model, both embedding table operations and dense computations take a significant fraction of overall training iteration time. Models with smaller dense networks will be more bottlenecked on embedding table operations, and models with fewer tables or with fewer lookups per output will be more compute bound.

The input distributions for embedding tables used in industry are proprietary, though literature suggest that there is at least some temporal locality. In this work, we chose to select two extremes. First, we use a uniform random input distribution for all tables. This is nearly the worst case for caching as there is limited reuse. Second, we use a Zipf [**79**] distribution with $\alpha = 1$ for each table, scrambling the input for each table using a maximum length LFSR starting at a random phase. This distribution has significant temporal locality. Dense inputs were generated using a normal distribution.

As a baseline, we ran the large network in *memory mode*, using CachedArrays for fast memory allocation.

5.6.3. Results. The results for out large DLRM model are shown in Figure 5.12. Figure 5.12a shows performance when a uniform distribution is used to drive sparse accesses while Figure 5.12b demonstrates the same model for the Zipf distribution.

5.6.3.1. *Understanding Uniform Results.* We first discuss the results for the uniform distribution, beginning by comparing the 2LM only run with *CachedArrays* simple. Between the two, *CachedArrays* simple is around 3.07× faster for embedding table lookups, despite using no DRAM for embedding table vectors. To understand this, we must characterize the steady-state of the DRAM cache as training progresses. Recall that our embedding tables greatly exceed the size of DRAM and are driven by a uniform distribution. In training, each row in the table that is accessed during the forward pass is updated on the backward pass and hence becomes dirty with respect to the DRAM cache. After sufficient iterations, the DRAM cache will become nearly completely dirty. The lookup phase will then trigger a large number of dirty writebacks. Combined with the random access behavior of embedding tables, the cache operates in a highly undesirable state, decreasing performance. However, in 2LM, feature vectors accessed during the lookup phase will be moved into the DRAM cache. Thus, gradient updates to these vectors will hit in the DRAM cache, accelerating the subsequent update phase to be 2.83× faster than the *simple* policy which always write to PM. Indeed, the timings of the lookup and update phase essentially swap between the 2LM and *simple* runs. In 2LM, the dirty cache also lowers performance of the dense layers due to spurious dirty misses, demonstrating that performance optimization in the presence of a DRAM cache cannot be performed component by component.

Now we discuss the three CachedEmbeddings based runs. Observe that for all three of these runs, the performance of the dense layers is nearly the same. This is expected since now all dense computations are performed with memory in DRAM. The *simple* case is capable of achieving nearly the whole bandwidth of the PM devices. However, since embedding table updates must be done directly into PM, we see a performance degradation due to the low PM write bandwidth. The *static* policy performs the best. In this mode, embedding table lookup and update operations are serviced from both DRAM and PM. Thus, there is a performance benefit if for accessing rows in DRAM over the *simple* policy without a performance loss if the vector is in PM. The *dynamic* policy is able to perform a little better than the *simple* one because all embedding table updates go to DRAM. However, it is slower then *static* for embedding table lookups because the eager caching of embedding table vectors incurring more DRAM write bandwidth, competing with PM reads.

Further more, *dynamic* incurs a slightly higher update penalty due to cache management (writing back dirty rows from old cache pages).

5.6.3.2. *Understanding Zipf Results.* When switching from a uniform distribution (low reuse) to a Zipf distribution ($\alpha = 1$, high reuse), we observe speedups in embedding table and lookup performance across the board. Several factors are at play here. First, with this level of reuse, CPU caches become effective, reducing overall memory traffic. The embedding table update sees further performance increases due to our gradient aggregation strategy where the entire gradient for each embedding table vector is accumulated before applying the optimizer. With higher reuse, there are fewer unique indices per lookup triggering lower write traffic to PM.

Finally, we can see the effect of 2LM and CachedEmbeddings based caching mechanisms. The lookup performance of 2LM increases by 4× as the DRAM cache stops experiencing such a high miss rate. Further, the performance of *dynamic* improves by 2.85× compared to with the uniform distribution, surpassing the static strategy since it is able to correctly cache the hot vectors in DRAM. Indeed, we observe that there is even a slight performance regression of *simple* when compared to 2LM as there is enough locality in the accessed vectors to overcome some of the issues associated with the hardware managed DRAM cache.

We again see the benefit of adding knowledge of program behavior to the memory management policy. When the sparse input distribution is uniform, our cache is too small to have a high enough hit rate to offset the overhead of moving vectors into the cache. In this case, a static partition of the data structures results in better utilization of the multiple levels of memory. However, when there *is* enough temporal locality in the input distribution for caching to be effective, fine grained memory management is exactly what we need. Tailoring of policy to the specifics of hardware and runtime situation is essential for performance.

5.7. Related Work

Bandana [**31**] aims to reduce the amount DRAM required for DLRM inference workloads on CPU clusters by using a combination of DRAM and SSDs, using heuristics to determine how to cache embedding vectors in DRAM. Like our work, Bandana also caches hot vectors in DRAM. However, Bandana needs to overcome the coarse read granularity of SSDs and must use hypergraph partitioning to group vectors with spatial locality to the same sector within the SSD. Persistent memory does not have this limitation, so this work investigates fine-grained vector caching while still maintaining high read and write bandwidth to PM.

There are two state-of-the-art implementations of DLRM systems in recent literature. Facebook's NEO [**66**] is software/hardware codesign of large scale DLRM models on a custom GPU-based hardware platform called ZionEX. It uses a customized 32-way set-associative software cache with LRU and LFU cache replacement policies and enables fine grain control of caching and replacement. Though NEO is focused on the GPU ecosystem, it provides motivation for the need of software managed caches to deal with large embedding tables. Intel's DLRM implementation [**52**] focuses on efficient parallelization across multiple CPU and a novel implementation of the SGD optimizer targeting mix-precision training. We extend this work by proposing a scale-up solution taking advantage of heterogeneous memory. We compare with Intel's implementation (Section 5.5), yielding a 1.4-2× speedup on the same hardware resources.

CHAPTER 6

Conclusions and Future Work

6.1. Limitations

The implementation of data tiering outlined in this work has many limitations. For one, the allocator used for *CachedArrays* would not scale well to facilitate even moderate levels of concurrent allocation and deallocation. Even if this were solved, there would be scalability issues regarding Julia's garbage collector, itself not very parallel friendly, which is required to track allocated object in order to call finalizers.

Furthermore, the applications studied in this thesis largely revolved around kernel-based programs. Heterogeneous memory management for these programs is relatively straightforward due to

```
# Function definition
julia> function thrash(x::CachedArray)
   Threads.@threads for tid in 1:2
       if tid == 1
           x .= 0
       elseif tid == 2
           CachedArrays.evict!(x)
       end
   end
end

julia> manager = DataManager(...);

julia> x = CachedArray(ones(Int64, 2_000_000_000), manager);

# All entries are one.
julia> sum(x)
2000000000

julia> thrash(x);
# Some entries got set to zero but not all.
julia> sum(x)
1749999992
```

Listing 6.1. Function showing synchronization issues that can occur when using *CachedArrays*. The function `thrash` launches two threads, the first zeros the passed array and the other evicts it. This results in a race condition where the array is evicted part-way through writing, resulting in a torn write.

regular synchronization. That is, while individual kernels like convolutions or dense inner-products are heavily parallelized, the time between kernel invocations is serial. This provides a natural synchronization point for memory management.

If, instead, a program is more concurrent with its memory management, issues can arise. This is demonstrated in Listing 6.1 which shows an instance of an `CachedArray` eviction during a write, resulting in an indeterminate result. Synchronization issues like this aren't new and indeed can show up in existing code using normal constructs like `std::vector` when the backing memory is reallocated concurrently with other accesses. The point here is that memory management in the context of concurrent access requires more careful synchronization.

A strategy to mitigate this is to use the `ReadOnly` and `ReadWrite` attributes to control array access. Upon conversion of a `CachedArray` to `ReadWrite` or `ReadOnly`, the policy could restrict any kind of eviction or prefetching of the array until the array is released back to a `NotBusy` state.

Since Julia is a relatively young language, there do not exist many state-of-the-art applications written solely within the language to serve as benchmarks for testing heterogeneous memory. We were able, through engineering effort, able to build somewhat competitive CNN and DLRM models, but these very much lack the actual robustness that would be required for production scale implementations. Possible future directions could involve using the Julia's differential equations[1] ecosystem for more workloads, though for the current incarnation of *CachedArrays*, this would necessitate focusing on problems that memory limited.

6.2. Hardware Support for Data Tiering

6.2.1. Data Movement Engines. Next generations of processors will contain dedicated memory movement engines such as Intel's data streaming accelerator (DSA) [**8**]. These accelerators serve as high performance memory copy, fill, and delta generation engines accessible from user-space and capable of operating on virtual addresses. One of the primary benefits behind such accelerators is offloading the task of data movement from the CPU cores. In the context of *CachedArrays*, this would be helpful for implementing the `copyto!` function in the data manager

[1]`https://diffeq.sciml.ai/stable/`

API. Currently, this function uses a multi-threaded memory copy to quickly move regions around so such accelerators could be helpful.

These accelerators could also be beneficial in the context of *CachedEmbeddings* as well. Movement of a feature vector in *CachedEmbeddings* requires the use of the AVX registers in the CPU. However, as discussed in Section 5.3, use of the AVX registers to hold partial products for reducing lookups is helpful for performance. Thus, movement of a vector part-way through a reducing lookup either can require spilling some registers to memory. It's unclear yet if the DSA will work well for small movement sizes, but if it does, than *CachedEmbeddings* style of memory management could benefit from this accelerator.

6.2.2. Support Via Virtual Memory. Perhaps one of the biggest limitations of the current *CachedArrays* implementation is its ability to efficiently support sub-object segmentation. As a motivating example, consider the case of using a very fast on-chip SRAM scratchpad to accelerate a computation (e.g., a linear algebra routine on a large matrix). If the matrices involved are sufficiently large such that they cannot fit within the SRAM cache in their entirety, they will need to be partitioned according to some strategy. *CachedArrays* cannot support transparent partitioning since an array is assumed to be contiguous in memory. Instead, either appropriately sized intermediate arrays will need to be allocated and explicitly used, or some higher-order array type constructed from multiple *CachedArrays* would have to be used. The latter is not ideal because it would involve multiple levels of indirection to access array elements and would result in the higher-order object no longer having a strided memory layout and thus unsuitable for BLAS libraries. An argument can be made for the former in that it requires programs to be explicit about data movement and scratchpad management, and won't necessarily require arrays to be aligned with page boundaries.

Now, modern computers already have support hardware accelerated indirect memory accesses through the virtual memory subsystem. That is, array-like objects can be contiguous in the virtual memory address space without being contiguous in physical address space. From a heterogeneous data-tiering standpoint, this can be used to partition a large contiguous array into multiple transparent objects (i.e., pages) which can be independently moved between different memory tiers. This would require rearchitecting the virtual memory subsystem to (1) support fast reassignment of virtual addresses to physical addresses, preferably without causing a TLB shootdown and its associated overheads and (2) preferable happen without incurring an expensive OS context switch. Both of these seem difficult from an implementation and coherence stand point.

Orthogonally, hardware can support concurrent data movement and access. As discussed previously, concurrent array access and movement in *CachedArrays* will result in a data race. This necessitates either global synchronization (as is the case in CNN training) or per-object locks/semaphores for correct behavior. Operating system/hardware support for data management through the virtual memory subsystem could provide mechanisms like allow trapping upon access to a migrating object, simplifying the user-level burden of synchronization.

6.2.3. Hardware Support for Policy Implementations. Another aspect in which hardware support can help efficient data management is in providing real time telemetry on address range usage. For example, policy mechanisms like `read_use` or `write_use` within *CachedArrays* can be used to communicate to the policy when the corresponding arrays are used. However, this cannot necessarily describe to the policy how *how heavily* a given array is accessed. For example, entries in array A may be accessed only once, or maybe sparsely, while entries in array B may be accessed many times with high temporal and spatial locality. Now, the policy may be able to indirectly infer usage with profiling techniques like those used in AutoTM, but this may require profiling overhead.

Instead, we can imagine hardware support for registering objects (i.e., address ranges) as regions of interest within the CPU core. All loads and stores to addresses within these objects can be accumulated in registers that can be queried by the policy to determine which objects are hotly accessed. This, in turn, can influence the policy's decision on how to handle these objects. We can take this even further and allow the policy to expose object usage statistics to the application to allow the application to provide better hints to the policy.

APPENDIX A

OneDNN Wrapper

A.1. OneDNN Summary

A software component critical to the later chapters in this work is oneDNN[1]. This is an Intel library offering high-performance implementations of many deep-learning primitives such as convolutions, dense-layers, batch normalization etc. This appendix will cover exposing the library's functionality to Julia.

A.1.1. Operations. OneDNN kernels are implemented as *primitives*, which are essentially type-erased implementations of deep-learning operations. Primitives first begin as *operation descriptors*, which include the type of operation to perform, the sizes and layouts of all arguments and output parameters, and other miscellaneous parameters like optional post-ops, constant values, etc. Operation descriptors are then used to construct *primitive descriptors.* Primitive descriptors take a provided execution backend and operation descriptor and will fill out any missing details from the operation descriptor. For example, primitive descriptors can determine the optimal layout of arguments, which can then be queried. Finally, primitive descriptors are used to construct *primitives*, which use just-in-time code generation along with all metadata stored in the primitive descriptor to generate a high performance implementation of the requested operation.

Primitives can then invoked with all input and output parameters.

A.1.2. Memory. Memory for primitive input and output parameters is supplied through the *memory* class. OneDNN supports a large number of exotic tiled memory formats to accelerate primitive operation. The format of a particular *memory* instance can be queried through its *memory descriptor*, which includes other relevant information like logic size and padded memory footprint. The actual pointer for the memory can come from either the oneDNN library or from the user application. This latter approach is used in this work.

[1]`https://github.com/oneapi-src/oneDNN`

```
# Before Macro Expansion
@apicall dnnl_primitive_get_primitive_desc(primitive, _pd)

# Post Macro Expansion
hummingbird = (OneDNN.Lib).dnnl_primitive_get_primitive_desc(
    OneDNN.dnnl_convert(primitive),
    OneDNN.dnnl_convert(_pd),
)
if hummingbird != (OneDNN.Lib).dnnl_success
    OneDNN.error("DNNL Failure: " * string(hummingbird))
end
hummingbird

# Default definition for `dnnl_convert`
dnnl_convert(x) = x
```

Listing A.1. Example lowering of the `@apicall` macro. The conversion helper `dnnl_convert` is called on each argument. By default, this function is a no-op. The return code is also checked for success or failure.

A.2. Exposing the C API

A.2.1. Generating FFI Functions. OneDNN has a C-interface composed of hundreds of functions. While Julia has native support for calling C, manually writing Julia for each C function is tedious and error prone. Instead, the Julia package Clang.jl[2] was used. This is a package that interfaces with Clang and LLVM to take C header files and automatically generate equivalent Julia types and wrapper functions for all C types and functions in the header.

A.2.2. C Calls and Type Conversion. The oneDNN library uses integer return types to denote failure or success of an operation. Additionally, some conversion may be required between Julia types and the corresponding oneDNN type for seamless use. To this end, the `@apicall` (shown in Listing A.1) macro was introduced for optional argument conversion and error handling.

In addition to `dnnl_convert`, Julia's normal `ccall` conversion (shown in Listing A.2) provides two more, `Base.cconvert` and `Base.unsafe_convert`. By default, the implementation of these functions is simple and often completely by the compiler. However, there are two illustrative cases where we can take advantage of this type conversion pipeline.

First, we would like to reconcile ABI issues with Julia's representation of `dnnl_dims_t`, the type that contains the logical dimensions of a tensor. On the C side, this is represented as

[2]`https://github.com/JuliaInterop/Clang.jl`

```julia
# Inside the C function wrapper
function dnnl_primitive_get_primitive_desc(primitive, primitive_desc)
    return ccall(
        (:dnnl_primitive_get_primitive_desc, libdnnl),
        dnnl_status_t,
        (const_dnnl_primitive_t, Ptr{const_dnnl_primitive_desc_t}),
        primitive,
        primitive_desc,
    )
end

# `ccall` argument conversion expansion
function dnnl_primitive_get_primitive_desc(primitive, primitive_desc)
    return ccall(
        (:dnnl_primitive_get_primitive_desc, libdnnl),
        dnnl_status_t,
        (dnnl_primitive_t, Ptr{dnnl_primitive_desc_t}),
        Base.unsafe_convert(dnnl_primitive_t, Base.cconvert(dnnl_primitive_t, primitive))
        Base.unsafe_convert(dnnl_primitive_desc_t, Base.cconvert(dnnl_primitive_desc_t,
    primitive_desc)),
    )
end
```

Listing A.2. Example of normal Julia lowering of `ccode`, providing two methods for potential conversion: `Base.cconvert` and `Base.unsafe_convert`. The result of `cconvert` is protected from early garbage collection (if applicable) while `unsafe_convert` is meant to handle conversion of pointer types.

```c
typedef dnnl_dims_t int64_t[12];
```

while on the Julia side this is:

```julia
const dnnl_dims_t = NTuple{12,Int64}
```

When calling C functions accepting `dnnl_dims_t`, the C function expects a pointer while Julia wants to pass the tuple on the stack. We can't change the Julia definition of `dnnl_dims_t` to an array because downstream bits types like `dnnl_memory_desc_t` have `dnnl_dims_t` as a member, so changing `dnnl_dims_t` 's definition would break all kinds of things. Furthermore, we can't redefine `Base.unsafe_convert` nor `Base.cconvert` because that would be type piracy and may have undefined consequences[3]. Instead, we can simply define:

```julia
dnnl_convert(x::NTuple{12,Int64}) = Ref(x)
```

[3]`https://docs.julialang.org/en/v1/manual/style-guide/#Avoid-type-piracy`

The function `Base.unsafe_convert` will then perform the correct conversion to a pointer and everything is happy.

Another example where we can use this conversion pipeline to our advantage is by implementing strict type checks for pointer conversions. By default, Julia is perfectly happy converting a pointer from one type to another. For example, the following is allowed, despite whether or not it makes sense:

```julia
ptrA = Ptr{Int}(0)
ptrB = Base.unsafe_convert(Ptr{NTuple{4,Float32}}, ptrA)
```

Many of the functions on the oneDNN C API take various types by pointers, which means that if this kind of free pointer conversion is allowed, then *when* we accidentally mess up the order of types passed to the `ccall`, then we get a rather unhelpful segfault instead of a more helpful error message. This is fixed by teaching Clang.jl's wrapping script to explicitly turn incorrect pointer conversion of oneDNN defined types into an error, as illustrated below.

```julia
struct dnnl_engine end
function Base.cconvert(::Type{Ptr{dnnl_engine}}, x::Ptr{dnnl_engine})
    return x
end
function Base.cconvert(::Type{Ptr{dnnl_engine}}, x::Ptr)
    error("Refusing to convert \$(typeof(x)) to a Ptr{\$(dnnl_engine)}!")
end
function Base.cconvert(::Type{Ptr{Ptr{dnnl_engine}}}, x::Ptr{Ptr{dnnl_engine}})
    return x
end
function Base.cconvert(::Type{Ptr{Ptr{dnnl_engine}}}, x::Ptr)
    msg = join((
        "Refusing to convert ",
        typeof(x),
        " to a Ptr{Ptr{dnnl_engine}}",
    ))
    error(msg)
end
```

Listing A.3. Automatically generated code to disable implicit pointer conversions for the `dnnl_engine` type. Similar conversion restrictions are used for all critical types exposed by the oneDNN C-API.

```julia
 1  mutable struct Memory
 2      handle::dnnl_memory_t
 3      Memory() = new(dnnl_memory_t())
 4  end
 5
 6  # Helper Constructors
 7  # Function `memorydesc` returns a oneDNN type that describes the memory layout
 8  # of the array `A`.
 9  function Memory(A::DenseArray, desc = memorydesc(A); kw...)
10      return Memory(convert(Ptr{Nothing}, pointer(A)), desc)
11  end
12
13  function Memory(ptr::Ptr{Nothing}, desc; engine = global_engine())
14      memory = Memory()
15      @apicall dnnl_memory_create(memory, desc, engine, ptr)
16      finalizer(memory) do _memory
17          @apicall dnnl_memory_destroy(_memory)
18      end
19      return memory
20  end
21
22  # Automatic conversion to C-types
23  Base.unsafe_convert(::Type{dnnl_memory_t}, x::Memory) = x.handle
24  function Base.unsafe_convert(::Type{Ptr{dnnl_memory_t}}, x::Memory)
25      return Base.unsafe_convert(Ptr{dnnl_memory_t}, Base.pointer_from_objref(x))
26  end
```

LISTING A.4. Exposing the `dnnl_memory` oneDNN type to Julia.

A.3. Exposing Types to Julia

The oneDNN C-API uses a large number of opaque struct pointers such as `dnnl_memory_t`. On the Julia side, this is easily modeled as

```julia
# Doesn't really matter if `dnnl_memory` is `mutable` or not since it is never
# actually materialized, we only deal with `Ptr{dnnl_memory}`.
struct dnnl_memory end
const dnnl_memory_t = Ptr{dnnl_memory}
```

Many of these types have cleanup code that must be called when the types are destroyed. The fix is straightforward: we introduce a mutable type wrapper for the various opaque pointers and use a finalizer to call the appropriate destructors. The pattern looks something like Listing A.4. The function doing the heavy lifting is on lines 13 to 20. This allocates the `dnnl_memory` pointer (line 14 - since `Memory` is a `mutable struct`, it is guaranteed to have stable memory address), create the actual memory object using the appropriate oneDNN C function (line 15), and attaches a finalizer to the allocated `Memory` that will clean up the C allocated data structures using the appropriate destructor (line 16-18). To allow for automatic C function type conversion, we define the conversion functions one lines 23 to 26, allowing for conversions of a `Memory` to a C `dnnl_memory*` and to a `dnnl_memory**`. This pattern of type wrapping is used for enough oneDNN types that a macro (`@wrap_type`) was created to automatically generate most of the boiler-plate.

A.3.1. Primitive Argument Pipeline. OneDNN's API for primitive arguments (i.e., source and destination tensors) involves passing a C array of `dnnl_exec_arg_t` whose Julia equivalent is defined as

```
struct dnnl_exec_arg_t
    arg::Cint
    memory::dnnl_memory_t
end
```

Where the `arg` field is essentially an enum describing the arguments role. For example, `DNNL_ARG_SRC_0` has a value of 1 while `DNNL_ARG_DST_0` has a value of 17. All such arguments are gathered into an array (or tuple) and passed to oneDNN as a pointer. When the number of arguments is small and known, it is more efficient to use a tuple since Julia's optimizer can elide any allocation and simply pass oneDNN a pointer to the variables on the stack.

As is the case with many of these oneDNN bridging functions, manually constructing the collection of arguments can be tedious and error prone. To that end, a macro "`@dnnl_args`" is introduced to perform the heavy lifting. An example usage is shown below

```
# Macro Invocation for
@dnnl_args src dst
@dnnl_args src scale_shift dst mean variance workspace
@dnnl_args dst multiple_src

# Expanded Macros
OneDNN.make_args(
    OneDNN.dnnl_arg((OneDNN.Lib).DNNL_ARG_SRC, src, OneDNN.Reading()),
    OneDNN.dnnl_arg((OneDNN.Lib).DNNL_ARG_DST, dst, OneDNN.Writing())),
)
OneDNN.make_args(
    OneDNN.dnnl_arg((OneDNN.Lib).DNNL_ARG_SRC, src, OneDNN.Reading()),
    OneDNN.dnnl_arg((OneDNN.Lib).DNNL_ARG_SCALE_SHIFT, scale_shift, OneDNN.Reading()),
    OneDNN.dnnl_arg((OneDNN.Lib).DNNL_ARG_DST, dst, OneDNN.Writing()),
    OneDNN.dnnl_arg((OneDNN.Lib).DNNL_ARG_MEAN, mean, OneDNN.Reading()),
    OneDNN.dnnl_arg((OneDNN.Lib).DNNL_ARG_VARIANCE, variance, OneDNN.Reading()),
    OneDNN.dnnl_arg((OneDNN.Lib).DNNL_ARG_WORKSPACE, workspace, OneDNN.Writing()),
)
OneDNN.make_args(
    OneDNN.dnnl_arg((OneDNN.Lib).DNNL_ARG_DST, dst, OneDNN.Writing()),
    OneDNN.dnnl_arg((OneDNN.Lib).DNNL_ARG_MULTIPLE_SRC, multiple_src, OneDNN.Reading()),
)
```

There are several things to note about the expanded macro. First, the name of the variable on the Julia side is used to find the correct integer value to describe that argument's role (e.g., `src` is converted into `OneDNN.Lib.DNNL_ARG_SRC`). This means that any code using this macro needs to keep its variable names in alignment with the oneDNN API, which is useful for code clarity as variable names on the Julia side will match the oneDNN documentation. Second, a table is kept regarding the usage of each argument type. For example, we know that any ”source” arguments will be read and any ”dst” arguments will be written. This can be used to provide context to argument creation (e.g., `OneDNN.Reading()`), which in turn allows packages like CachedArrays.jl to record read and write accesses to the memory buffers backing the `dnnl_memory_t` objects.

A.4. Putting it all Together

An example implementation of elementwise operations is shown in Listing A.5. This uses much of the machinery developed previously to build and execute kernels performing an operation on each element of a tensor (exposed as the `Memory` type in Julia).

Note that creation of the primitive from the primitive descriptor (handled by the `temp_primitive` function, results in JIT compiling the kernel. The compiled kernel implementation is specialized on most aspects of the kernel, including element type, tensor dimensions, and tensor layout. To avoid recompiling the primitive implementation each time, the oneDNN library maintains a primitive cache, using the primitive descriptor as key for this cache. Thus, there is a small overhead of accessing the cached primitive implementation, but this is usually negligible compared to the overall execution time of the primitive.

Through these means, we were able to expose the functionality of oneDNN to implement high-performance deep neural networks. Furthermore, because memory allocation was controlled on the Julia side, *CachedArrays* could be used as the memory allocator for oneDNN *memory* types. This provides full control of the location (i.e., DRAM or PM) of each kernel's input and output parameters, allowing for heterogeneous memory management of oneDNN operations.

```julia
# Map Julia functions to OneDNN `eltwise` arguments.
forward_expand(::typeof(Base.abs)) = (Lib.dnnl_eltwise_abs, zero(Float32), zero(Float32))
function forward_expand(::typeof(Flux.sigmoid))     ■ Missing reference: sigmoid
  return (Lib.dnnl_eltwise_logistic, zero(Float32), zero(Float32))
end
forward_expand(::typeof(Base.sqrt)) = (Lib.dnnl_eltwise_sqrt, zero(Float32), zero(Float32))
forward_expand(::typeof(Flux.relu)) = (Lib.dnnl_eltwise_relu, zero(Float32), zero(Float32))
forward_expand(::typeof(Base.log)) = (Lib.dnnl_eltwise_log, zero(Float32), zero(Float32))

# Implement `eltwise~.
eltwise(f::F, src::Memory) where {F} = eltwise(src, forward_expand(f)...)
eltwise(::typeof(identity), src::Memory) = src
function eltwise(
    src::Memory, kind::dnnl_alg_kind_t, alpha = one(Float32), beta = zero(Float32)
)
    # Keep similar format to source
    dst = similar(src)
    eltwise!(dst, src, kind, alpha, beta)
    return dst
end

function eltwise!(
    dst::Memory,
    src::Memory,
    algo::Lib.dnnl_alg_kind_t,
    alpha = one(Float32),
    beta = zero(Float32),
)
    # Create an operation descriptor.
    opdesc = Ref{Lib.dnnl_eltwise_desc_t}()
    @apicall dnnl_eltwise_forward_desc_init(opdesc, Inference(), algo, src, alpha, beta)

    # From the operation descriptor, create a primitive descriptor and then the primitive.
    temp_primitive(opdesc, noattributes(), global_engine(), noforward()) do primitive, _
        execute!(primitive, @dnnl_args dst src)
    end
    return dst
end
```

Listing A.5. Implementation of elementwise operations. Methods `forward_expand` turn Julia functions into triplets including the oneDNN operation enum and scaling factors alpha and beta. The function `eltwise` takes a `Memory` (a wrapper type for oneDNN's tensor type with memory allocated by Julia), constructs a similar sized destination for the operation using `similar`, and calls the mutating `eltwise!`. Inside the implementation of `eltwise`, first an operation descriptor `opdesc` is created. This is then passed to a function `temp_primitive`, which create a primitive descriptor and finally the primitive itself. Both of these objects are passed to the body of the `temp_primitive`'s closure. With the primitive, oneDNN arguments are created (`@dnnl_args`) and the `execute!` is called. This function is responsible for any required scratchpad allocation and finally executes the primitive.

Bibliography

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, *Tensorflow: A system for large-scale machine learning*, in 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), Savannah, GA, 2016, USENIX Association, pp. 265–283.

[2] N. Agarwal and T. F. Wenisch, *Thermostat: Application-transparent page management for two-tiered main memory*, in Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017, 2017, pp. 631–644.

[3] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, *On subnormal floating point and abnormal timing*, in Proceedings of the 2015 IEEE Symposium on Security and Privacy, SP '15, Washington, DC, USA, 2015, IEEE Computer Society, pp. 623–639.

[4] O. Avissar, R. Barua, and D. Stewart, *Heterogeneous memory management for embedded systems*, in Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES '01, New York, NY, USA, 2001, ACM, pp. 34–43.

[5] M. Bartholomew-Biggs, S. Brown, B. Christianson, and L. Dixon, *Automatic differentiation of algorithms*, Journal of Computational and Applied Mathematics, 124 (2000), pp. 171–190. Numerical Analysis 2000. Vol. IV: Optimization and Nonlinear Equations.

[6] S. Beamer, *Understanding and improving graph algorithm performance*, PhD thesis, UC Berkeley, 2016.

[7] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, *Julia: A fresh approach to numerical computing*, SIAM review, 59 (2017), pp. 65–98.

[8] A. Biswas, *Sapphire rapids*, in 2021 IEEE Hot Chips 33 Symposium (HCS), 2021, pp. 1–22.

[9] A. Brock, J. Donahue, and K. Simonyan, *Large scale GAN training for high fidelity natural image synthesis*, in 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019, 2019.

[10] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess,

J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, *Language models are few-shot learners*, 2020.

[11] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli, *Rethinking software runtimes for disaggregated memory*, in Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2021, pp. 79–92.

[12] E. Carson, J. Demmel, L. Grigori, N. Knight, P. Koanantakool, O. Schwartz, and H. V. Simhadri, *Write-avoiding algorithms*, in 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2016, pp. 648–658.

[13] C. L. Chen and M. Y. Hsiao, *Error-correcting codes for semiconductor memory applications: A state-of-the-art review*, IBM Journal of Research and Development, 28 (1984), pp. 124–134.

[14] H. Chen, W. Chen, J. Huang, B. Robert, and H. Kuhn, *MPIPP: an automatic profile-guided parallel process placement toolset for SMP clusters and multiclusters*, in Proceedings of the 20th Annual International Conference on Supercomputing, ICS 2006, Cairns, Queensland, Australia, June 28 - July 01, 2006, 2006, pp. 353–360.

[15] X. Chen, D. Z. Chen, and X. S. Hu, *modnn: Memory optimal dnn training on gpus*, in 2018 Design, Automation Test in Europe Conference Exhibition (DATE), March 2018, pp. 13–18.

[16] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, *cudnn: Efficient primitives for deep learning*, CoRR, abs/1410.0759 (2014).

[17] C. Chou, A. Jaleel, and M. K. Qureshi, *Bear: Techniques for mitigating bandwidth bloat in gigascale dram caches*, in 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA), 2015, pp. 198–210.

[18] C. C. Chou, A. Jaleel, and M. K. Qureshi, *Cameo: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache*, in 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, 2014, pp. 1–12.

[19] R. Collobert and J. Weston, *A unified architecture for natural language processing: Deep neural networks with multitask learning*, in Proceedings of the 25th international conference on Machine learning, ACM, 2008, pp. 160–167.

[20] T. Cormen, C. Leiserson, R. Rivest, and e. Clifford Stein, *Introduction to Algorithms*, MIT Press, 2001.

[21] I. Corporation, *Optane dc persistent memory brief*.

[22] ———, *Intel 64 and IA-32 Architectures Software Developer's Manual*, Intel Corporation, August 2016.

[23] ———, *onednn*. `https://github.com/oneapi-src/oneDNN`, 2021.

[24] S. Cyphers, A. K. Bansal, A. Bhiwandiwalla, J. Bobba, M. Brookhart, A. Chakraborty, W. Constable, C. Convey, L. Cook, O. Kanawi, R. Kimball, J. Knight, N. Korovaiko, V. Kumar, Y. Lao,

C. R. Lishka, J. Menon, J. Myers, S. A. Narayana, A. Procter, and T. J. Webb, *Intel ngraph: An intermediate representation, compiler, and executor for deep learning*, CoRR, abs/1801.08058 (2018).

[25] N. S. Dasari, R. Desh, and M. Zubair, *Park: An efficient algorithm for k-core decomposition on multicore processors*, in 2014 IEEE International Conference on Big Data (Big Data), 2014, pp. 9–16.

[26] L. Dhulipala, C. McGuffey, H. Kang, Y. Gu, G. E. Blelloch, P. B. Gibbons, and J. Shun, *Sage: Parallel semi-asymmetric graph algorithms for nvrams*, Proc. VLDB Endow., 13 (2020), p. 1598–1613.

[27] T. D. Doudali, D. Zahka, and A. Gavrilovska, *Cori: Dancing to the right beat of periodic data movements over hybrid memory systems*, in 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2021, pp. 350–359.

[28] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan, *Data tiering in heterogeneous memory systems*, in Proceedings of the Eleventh European Conference on Computer Systems, 2016, pp. 1–16.

[29] I. Dunning, J. Huchette, and M. Lubin, *Jump: A modeling language for mathematical optimization*, SIAM Review, 59 (2017), pp. 295–320.

[30] A. Eisenman, D. Gardner, I. AbdelRahman, J. Axboe, S. Dong, K. M. Hazelwood, C. Petersen, A. Cidon, and S. Katti, *Reducing DRAM footprint with NVM in facebook*, in Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018, 2018, pp. 42:1–42:13.

[31] A. Eisenman, M. Naumov, D. Gardner, M. Smelyanskiy, S. Pupyrev, K. M. Hazelwood, A. Cidon, and S. Katti, *Bandana: Using non-volatile memory for storing deep learning models*, CoRR, abs/1811.05922 (2018).

[32] B. Falsafi and D. A. Wood, *Reactive numa: A design for unifying s-coma and cc-numa*, SIGARCH Comput. Archit. News, 25 (1997), p. 229–240.

[33] G. Gill, R. Dathathri, L. Hoang, R. Peri, and K. Pingali, *Single machine graph analytics on massive datasets using intel optane dc persistent memory*, Proc. VLDB Endow., 13 (2020), p. 1304–1318.

[34] GitHub, *Graph500*. **https://github.com/graph500/graph500**, 2019.

[35] A. Goldberg, E. Tardos, and R. Tarjan, *Network flow algorithms*, (1989), p. 80.

[36] D. W. Goodwin and K. D. Wilken, *Optimal and near-optimal global register allocations using 0&ndash;1 integer programming*, Softw. Pract. Exper., 26 (1996), pp. 929–965.

[37] U. Gupta, X. Wang, M. Naumov, C. Wu, B. Reagen, D. Brooks, B. Cottel, K. M. Hazelwood, B. Jia, H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang, *The architectural implications of facebook's dnn-based personalized recommendation*, CoRR, abs/1906.03109 (2019).

[38] L. Gurobi Optimization, *Gurobi optimizer reference manual*, 2018.

[39] F. T. Hady, A. Foong, B. Veal, and D. Williams, *Platform storage performance with 3D XPoint technology*, Proceedings of the IEEE, 105 (2017), pp. 1822–1833.

[40] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S. Liao, E. Bugnion, and M. S. Lam, *Maximizing multiprocessor performance with the SUIF compiler*, Digital Technical Journal, 10 (1998), pp. 71–80.

[41] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, CoRR, abs/1512.03385 (2015).

[42] J. Hestness, N. Ardalani, and G. Diamos, *Beyond human-level accuracy: Computational challenges in deep learning*, in Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, PPoPP '19, New York, NY, USA, 2019, ACM, pp. 1–14.

[43] M. Hildebrand, J. T. Angeles, J. Lowe-Power, and V. Akella, *A case against hardware managed dram caches for nvram based systems*, in 2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2021, pp. 194–204.

[44] M. Hildebrand, J. Khan, S. Trika, J. Lowe-Power, and V. Akella, *Autotm: Automatic tensor movement in heterogeneous memory systems using integer linear programming*, in Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '20, New York, NY, USA, 2020, Association for Computing Machinery, p. 875–890.

[45] T. Hirofuchi and R. Takano, *The preliminary evaluation of a hypervisor-based virtualization mechanism for intel optane DC persistent memory module*, CoRR, abs/1907.12014 (2019).

[46] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, *Densely connected convolutional networks*, in 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017, pp. 2261–2269.

[47] S. Ioffe and C. Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, in Proceedings of the 32nd International Conference on Machine Learning, F. Bach and D. Blei, eds., vol. 37 of Proceedings of Machine Learning Research, Lille, France, 07–09 Jul 2015, PMLR, pp. 448–456.

[48] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, *Basic performance measurements of the intel optane DC persistent memory module*, CoRR, abs/1903.05714 (2019).

[49] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, *Automatic CPU-GPU communication management and optimization*, in Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011, 2011, pp. 142–151.

[50] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, *Unison cache: A scalable and effective die-stacked dram cache*, in 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, 2014, pp. 25–37.

[51] D. Jevdjic, S. Volos, and B. Falsafi, *Die-stacked dram caches for servers: Hit ratio, latency, or bandwidth? have it all with footprint cache*, SIGARCH Comput. Archit. News, 41 (2013), p. 404–415.

[52] D. D. Kalamkar, E. Georganas, S. Srinivasan, J. Chen, M. Shiryaev, and A. Heinecke, *Optimizing deep learning recommender systems' training on CPU cluster architectures*, CoRR, abs/2005.04680 (2020).

[53] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan, *Heteroos: Os design for heterogeneous memory management in datacenter*, in Proceedings of the 44th Annual International Symposium on Computer Architecture, 2017, pp. 521–534.

[54] S. Kannan, Y. Ren, and A. Bhattacharjee, *Klocs: kernel-level object contexts for heterogeneous memory systems*, in Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2021, pp. 65–78.

[55] J. Kim, W. Choe, and J. Ahn, *Exploring the design space of page management for multi-tiered memory systems*, in 2021 {USENIX} Annual Technical Conference ({USENIX}{ATC} 21), 2021, pp. 715–728.

[56] C. Lameter, *Numa (non-uniform memory access): An overview*, Queue, 11 (2013), pp. 40:40–40:51.

[57] Y. LeCun, Y. Bengio, and G. Hinton, *Deep learning*, nature, 521 (2015), p. 436.

[58] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, *Gradient-based learning applied to document recognition*, Proceedings of the IEEE, 86 (1998), pp. 2278–2324.

[59] Y. Lee, J. Kim, H. Jang, H. Yang, J. Kim, J. Jeong, and J. W. Lee, *A fully associative, tagless dram cache*, in Proceedings of the 42nd Annual International Symposium on Computer Architecture, ISCA '15, New York, NY, USA, 2015, Association for Computing Machinery, p. 211–222.

[60] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, *Kronecker graphs: An approach to modeling networks*, J. Mach. Learn. Res., 11 (2010), p. 985–1042.

[61] G. H. Loh and M. D. Hill, *Efficiently enabling conventional block sizes for very large die-stacked dram caches*, in 2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2011, pp. 454–464.

[62] J. Lowe-Power, *On Heterogeneous Compute and Memory Systems*, PhD thesis, University of Wisconsin, Madison, 2017.

[63] P. Mattson, C. Cheng, C. Coleman, G. Diamos, P. Micikevicius, D. Patterson, H. Tang, G.-Y. Wei, P. Bailis, V. Bittorf, D. Brooks, D. Chen, D. Dutta, U. Gupta, K. Hazelwood, A. Hock, X. Huang, A. Ike, B. Jia, D. Kang, D. Kanter, N. Kumar, J. Liao, G. Ma, D. Narayanan, T. Oguntebi, G. Pekhimenko, L. Pentecost, V. J. Reddi, T. Robie, T. S. John, T. Tabaru, C.-J. Wu, L. Xu, M. Yamazaki, C. Young, and M. Zaharia, *Mlperf training benchmark*, 2019.

[64] S. McCandlish, J. Kaplan, D. Amodei, and O. D. Team, *An empirical model of large-batch training*, 2018.

[65] T. Mikolov, K. Chen, G. Corrado, and J. Dean, *Efficient estimation of word representations in vector space*, 2013.

[66] D. Mudigere, Y. Hao, J. Huang, Z. Jia, A. Tulloch, S. Sridharan, X. Liu, M. Ozdal, J. Nie, J. Park, L. Luo, J. A. Yang, L. Gao, D. Ivchenko, A. Basant, Y. Hu, J. Yang, E. K. Ardestani, X. Wang, R. Komuravelli, C.-H. Chu, S. Yilmaz, H. Li, J. Qian, Z. Feng, Y. Ma, J. Yang, E. Wen, H. Li,

L. Yang, C. Sun, W. Zhao, D. Melts, K. Dhulipala, K. Kishore, T. Graf, A. Eisenman, K. K. Matam, A. Gangidi, G. J. Chen, M. Krishnan, A. Nayak, K. Nair, B. Muthiah, M. khorashadi, P. Bhattacharya, P. Lapukhov, M. Naumov, A. Mathews, L. Qiao, M. Smelyanskiy, B. Jia, and V. Rao, *Software-hardware co-design for fast and scalable training of deep learning recommendation models*, 2021.

[67] M. Naumov, D. Mudigere, H. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy, *Deep learning recommendation model for personalization and recommendation systems*, CoRR, abs/1906.00091 (2019).

[68] D. Nguyen, A. Lenharth, and K. Pingali, *A lightweigth infrastructure for graph analytics*, in Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13, New York, NY, USA, 2013, Association for Computing Machinery, pp. 456–471.

[69] J. Nickolls, I. Buck, M. Garland, and K. Skadron, *Scalable parallel programming with cuda*, Queue, 6 (2008), pp. 40–53.

[70] T. Nowatzki, M. Ferris, K. Sankaralingam, C. Estan, N. Vaish, and D. Wood, *Optimization and mathematical modeling in computer architecture*, Synthesis Lectures on Computer Architecture, 8 (2013), pp. 1–144.

[71] I. Oukid, D. Booss, A. Lespinasse, W. Lehner, T. Willhalm, and G. Gomes, *Memory management techniques for large-scale persistent-main-memory systems*, Proceedings of the VLDB Endowment, 10 (2017), pp. 1166–1177.

[72] A. Outman, *Web data commons - hyperlink graphs*, tech. rep., 2017.

[73] L. Page, S. Brin, R. Motwani, and T. Winograd, *The pagerank citation ranking: Bringing order to the web.*, Technical Report 1999-66, Stanford InfoLab, November 1999.

[74] S. Pai, R. Govindarajan, and M. J. Thazhuthaveetil, *Fast and efficient automatic memory management for gpus using compiler-assisted runtime coherence scheme*, in International Conference on Parallel Architectures and Compilation Techniques, PACT '12, Minneapolis, MN, USA - September 19 - 23, 2012, 2012, pp. 33–42.

[75] W. Pan, T. Xie, and X. Song, *Hart: A concurrent hash-assisted radix tree for dram-pm hybrid memory systems*, in 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2019, pp. 921–931.

[76] O. Patil, L. Ionkov, J. Lee, F. Mueller, and M. Lang, *Performance characterization of a dram-nvm hybrid memory architecture for hpc applications using intel optane dc persistent memory modules*, in Proceedings of the International Symposium on Memory Systems, MEMSYS '19, New York, NY, USA, 2019, Association for Computing Machinery, p. 288–303.

[77] I. B. Peng, M. B. Gokhale, and E. W. Green, *System evaluation of the intel optane byte-addressable nvm*, in Proceedings of the International Symposium on Memory Systems, 2019, pp. 304–315.

[78] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe, *Portable performance on heterogeneous architectures*, in Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, New York, NY, USA, 2013, ACM, pp. 431–444.

[79] D. M. W. Powers, *Applications and explanations of Zipf's law*, in New Methods in Language Processing and Computational Natural Language Learning, 1998.

[80] M. K. Qureshi and G. H. Loh, *Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design*, in 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, 2012, pp. 235–246.

[81] A. Raad, J. Wickerson, G. Neiger, and V. Vafeiadis, *Persistency semantics of the intel-x86 architecture*, Proc. ACM Program. Lang., 4 (2019).

[82] M. Radulovic, D. Zivanovic, D. Ruiz, B. R. de Supinski, S. A. McKee, P. Radojković, and E. Ayguadé, *Another trip to the wall: How much will stacked dram benefit hpc?*, in Proceedings of the 2015 International Symposium on Memory Systems, MEMSYS '15, New York, NY, USA, 2015, ACM, pp. 31–36.

[83] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, *Zero: Memory optimizations toward training trillion parameter models*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '20, IEEE Press, 2020.

[84] R. K. Ramanujan, R. Agarwal, and G. J. Hinton, *Apparatus and method for implementing a multi-level memory hierarchy having different operating modes*, Feb. 2 2017.

[85] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, *Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters*, in Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery; Data Mining, KDD '20, New York, NY, USA, 2020, Association for Computing Machinery, p. 3505–3506.

[86] A. Raybuck, T. Stamler, W. Zhang, M. Erez, and S. Peter, *Hemem: Scalable tiered memory management for big data applications and real nvm*, in Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM, 2021, pp. 392–407.

[87] J. Ren, J. Luo, K. Wu, M. Zhang, H. Jeon, and D. Li, *Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning*, in 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA), IEEE, 2021, pp. 598–611.

[88] J. Revels, V. Churavy, S. Schaub, T. Besard, L. White, S. Kadowaki, M. J. Innes, T. Koolen, N. Daly, C. de Graaf, D. Aluthge, K. Fischer, K. Carlsson, M. Schauer, M. Piibeleht, R. Deits, and Rogerluo, *Julialabs/cassette.jl: v0.3.9*, Sept. 2021.

[89] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, *vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design*, in The 49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-49, Piscataway, NJ, USA, 2016, IEEE Press, pp. 18:1–18:13.

[90] A. Rohan, B. Panda, and P. Agarwal, *Reverse engineering the stream prefetcher for profit*, in 2020 IEEE European Symposium on Security and Privacy Workshops (EuroS PW), 2020, pp. 682–687.

[91] A. Schrijver, *Theory of Linear and Integer Programming*, John Wiley & Sons, Inc., New York, NY, USA, 1986.

[92] A. Shanbhag, N. Tatbul, D. Cohen, and S. Madden, *Large-scale in-memory analytics on intel® optane™ dc persistent memory*, in Proceedings of the 16th International Workshop on Data Management on New Hardware, DaMoN '20, New York, NY, USA, 2020, Association for Computing Machinery.

[93] L. G. Shapiro, *Connected component labeling and adjacency graph construction*, in Topological Algorithms for Digital Image Processing, T. Y. Kong and A. Rosenfeld, eds., vol. 19 of Machine Intelligence and Pattern Recognition, North-Holland, 1996, pp. 1 – 30.

[94] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. V. Le, G. E. Hinton, and J. Dean, *Outrageously large neural networks: The sparsely-gated mixture-of-experts layer*, CoRR, abs/1701.06538 (2017).

[95] Y. Shen and Z. Zou, *Efficient subgraph matching on non-volatile memory*, in International Conference on Web Information Systems Engineering, Springer, 2017, pp. 457–471.

[96] Y. Shiloach and U. Vishkin, *An o(logn) parallel connectivity algorithm*, Journal of Algorithms, 3 (1982), pp. 57 – 67.

[97] K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, in International Conference on Learning Representations, 2015.

[98] S. Smith, M. Patwary, B. Norick, P. LeGresley, S. Rajbhandari, J. Casper, Z. Liu, S. Prabhumoye, G. Zerveas, V. Korthikanti, E. Zhang, R. Child, R. Y. Aminabadi, J. Bernauer, X. Song, M. Shoeybi, Y. He, M. Houston, S. Tiwary, and B. Catanzaro, *Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model*, 2022.

[99] I. Sutskever, O. Vinyals, and Q. V. Le, *Sequence to sequence learning with neural networks*, in Advances in neural information processing systems, 2014, pp. 3104–3112.

[100] C. Szegedy, S. Ioffe, and V. Vanhoucke, *Inception-v4, inception-resnet and the impact of residual connections on learning*, CoRR, abs/1602.07261 (2016).

[101] S. Van Doren, *Abstract - hoti 2019: Compute express link*, in 2019 IEEE Symposium on High-Performance Interconnects (HOTI), 2019, pp. 18–18.

[102] A. van Renen, L. Vogel, V. Leis, T. Neumann, and A. Kemper, *Persistent memory i/o primitives*, in Proceedings of the 15th International Workshop on Data Management on New Hardware, 2019, pp. 1–7.

[103] A. van Renen, L. Vogel, V. Leis, T. Neumann, and A. Kemper, *Persistent memory I/O primitives*, CoRR, abs/1904.01614 (2019).

[104] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, *Attention is all you need*, in Advances in Neural Information Processing Systems, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds., vol. 30, Curran Associates, Inc., 2017.

[105] O. Vinyals, I. Babuschkin, J. Chung, M. Mathieu, M. Jaderberg, W. M. Czarnecki, A. Dudzik, A. Huang, P. Georgiev, R. Powell, et al., *Alphastar: Mastering the real-time strategy game starcraft ii*, DeepMind Blog, (2019).

[106] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska, *Superneurons: Dynamic GPU memory management for training deep neural networks*, CoRR, abs/1801.04380 (2018).

[107] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhano, *Characterizing and modeling non-volatile memory systems*, in IEEE/ACM International Symposium on Microarchitecture (MICRO), 2020.

[108] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, *Nimble page management for tiered memory systems*, in Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019, 2019, pp. 331–345.

[109] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, *An empirical guide to the behavior and use of scalable persistent memory*, in 18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020, 2020, pp. 169–182.

[110] H. Ze, A. Senior, and M. Schuster, *Statistical parametric speech synthesis using deep neural networks*, in 2013 ieee international conference on acoustics, speech and signal processing, IEEE, 2013, pp. 7962–7966.

[111] H. Zhang, T. Xu, H. Li, S. Zhang, X. Wang, X. Huang, and D. N. Metaxas, *Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks*, in Proceedings of the IEEE International Conference on Computer Vision, 2017, pp. 5907–5915.

[112] R. Zhou and T. M. Jones, *Janus: Statically-driven and profile-guided automatic dynamic binary parallelisation*, in Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Piscataway, NJ, USA, 2019, IEEE Press, pp. 15–25.